NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**A Computer Simulation of
Vehicle and Actuator Dynamics
for a Hexapod Walking Robot**

by

Karl Johann Ragnar Wrussell Kristiansen VII

March 1994

Thesis Advisor: Robert B. McGhee

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE
A Computer Simulation of Vehicle and Actuator Dynamics for a Hexapod Walking Robot (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Kristiansen VII, Karl Johann Ragnar Wrussell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

Underwater walking machines offer a potential for replacement of human divers in certain aspects of underwater construction and inspection. One such vehicle, Aquarobot, is currently under test in Japan. However, this vehicle is currently too slow to be economically utilized, and limited hardware availability restricts progress in control software improvements. A software dynamic simulation model is desirable to relieve this restricted access. The problem addressed by this research is the modeling of system dynamics of underwater walking vehicles with sufficient simplification to achieve a real-time simulation. The approach taken includes an object-oriented, massless leg robot dynamic model and employs a high performance graphics rendering toolkit.

The resulting simulations of a robotic joint actuator and of the robot itself, utilizing springs and dampers in the joints, runs in real-time. The robot simulation model executes on a four-processor machine with under fifteen percent utilization of the processor dedicated to system dynamics. This result indicates that the simulation is likely to retain real-time capability after replacing the springs and dampers with the more accurate joint actuator model also developed in this thesis.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Robotics, Aquarobot	165
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Approved for public release; distribution is unlimited

A COMPUTER SIMULATION OF VEHICLE AND ACTUATOR DYNAMICS FOR A HEXAPOD WALKING ROBOT

by

Karl Johann Ragnar Wrussell Kristiansen VII
Lieutenant, United States Navy
B.A., Lake Forest College, 1985

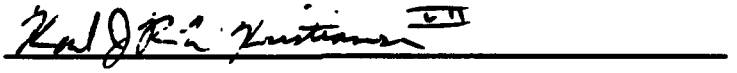
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1994

Author: 

Karl Johann Ragnar Wrussell Kristiansen VII

Approved By: 

Robert B. McGhee, Thesis Advisor

David R. Pratt, Second Reader

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Underwater walking machines offer a potential for replacement of human divers in certain aspects of underwater construction and inspection. One such vehicle, Aquarobot, is currently under test in Japan. However, this vehicle is currently too slow to be economically utilized, and limited hardware availability restricts progress in control software improvements. A software dynamic simulation model is desirable to relieve this restricted access. The problem addressed by this research is the modeling of system dynamics of underwater walking vehicles with sufficient simplification to achieve a real-time simulation. The approach taken includes an object-oriented, massless leg robot dynamic model and employs a high performance graphics rendering toolkit.

The resulting simulations of a robotic joint actuator and of the robot itself, utilizing springs and dampers in the joints, runs in real-time. The robot simulation model executes on a four-processor machine with under fifteen percent utilization of the processor dedicated to system dynamics. This result indicates that the simulation is likely to retain real-time capability after replacing the springs and dampers with the more accurate joint actuator model also developed in this thesis.

TABLE OF CONTENTS

I. INTRODUCTION

A. AQUAROBOT

Aquarobot is a six-legged "insect type", articulated, experimental prototype robot under development at the Port and Harbour Research Institute (PHRI) in Japan. This robot is being investigated as an alternative to the currently used human divers for seawall construction and inspection. While the divers are fully capable, the limited stay time at required depths make progress slow and expensive. A walking robot is preferred to tracked, wheeled, or floating versions for its abilities to maneuver without disturbing the bottom enough to cloud the water and restrict visibility, and to provide a stable reference platform from which measurements can be made [Iwasaki, 1987]. The prototype Aquarobot has successfully walked underwater and demonstrated functional feasibility for the intended task, but it is currently too slow to be economically utilized [Davidson, 1993]. The Naval Postgraduate School (NPS) is working with PHRI to upgrade Aquarobot's control software, from the original version written in BASIC, to improve its operating speed.

B. GOALS

The goal of this thesis is to investigate the feasibility of dynamic modeling of Aquarobot with sufficient simplifications to achieve a real-time simulation. The simulation

model should be statically accurate and dynamically approximate. The major simplifications considered are:

(1) massless legs,

(2) body mass evenly distributed in a cylinder,

(3) center of mass at geometric center of the inboard leg joints,

(4) infinite friction for foot contact with surface,

The dynamic Aquarobot model of this thesis uses springs and dampers in place of joint actuators for this initial feasibility study. A joint actuator simulation model, including servomotor and controller models, is also developed and is intended to eventually replace the springs and dampers. Inputs to that model will be control software orders to the joint motor controller.

C. ORGANIZATION

Chapter II of this thesis reviews previous and concurrent work in the area of walking robots with emphasis on work related to Aquarobot. Chapter III provides a more detailed description of Aquarobot and introduces the software tools used. Chapters IV and V develop the necessary mathematical models and then present prototype dynamic simulation models for an Aquarobot joint actuator and for Aquarobot itself (with springs and dampers in place of joint actuators). Chapter VI reviews the results of simulations accomplished with the models introduced in Chapters IV and V. Finally, Chapter VII presents some conclusions, suggestions for further research, and a summary.

II. SURVEY OF PREVIOUS AND CONCURRENT WORK

A. INTRODUCTION

To place Aquarobot research in relative perspective, this chapter begins with a brief historical review of walking machines. An overview of ongoing Aquarobot research at NPS follows and places this thesis in context. Other contributions, some completed and some currently in progress, are described. Also, as this thesis is a continuation of previous work, a short review of some of the key elements of that work is presented to provide a starting frame of reference.

B. BRIEF HISTORY OF WALKING MACHINES

In an early exploration of walking mechanisms, 1965 to 1968, General Electric Corporation built a four legged vehicle called the *Quadruped Transporter*. Because of the lack of theory and technology, designers incorporated human sensing and neural control of the limbs by attaching "position following, force feedback" control levers to the operator's arms and legs. Each of these levers had three degrees of freedom, corresponding to those of the leg it controlled. Very few mastered the skills required to operate the vehicle, and those that did found it to be very demanding [McGhee, 1985]. While the Quadruped Transporter lacked practicality, its successful implementation encouraged further research.

Automation of low level tasks, such as individual limb control, leaves the operator free to concentrate on higher level, "supervisory control" of the vehicle. In 1977, this method of control was used in the Ohio State University (OSU) "Hexapod Vehicle." The operator controlled vehicle speed and direction, using a joystick, while limb motion control and coordination was handled by computer. This machine was utilized in the development of gait algorithms [McGhee, 1985].

[McGhee, 1986] addresses the energy-efficiency issue of limb control and introduces a method used to achieve a cyclic leg motion without requiring the reversal of drive motors. This approach was demonstrated in MELCRABs 1 and 2 at Mechanical Engineering Laboratory in Japan.

From 1981 to 1986, the Adaptive Suspension Vehicle (ASV) was constructed and tested at OSU. The ASV was a six-legged vehicle designed for outdoor operation on irregular, unmapped terrain and included a self contained, onboard power supply. It carried an operator who exercised supervisory control via a joystick and keypad. Leg coordination and foothold selection were fully automated; the latter was allowed by employment of extensive environmental sensors including an optical terrain scanner [Waldron, 1986]. Related later work [Kwak, 1990] explores the use of rule-based limb motion coordination to implement a "free," non-periodic, gait permitting on-line optimization of foot placement.

C. AQUAROBOT RESEARCH AT NPS

Aquarobot research at NPS is divided into two concurrent phases: control and simulation. The first of these, control, consists of Aquarobot control software development. The second phase, simulation, involves development of a graphical computer simulation of the Aquarobot hardware. The simulator is required for the final stages of the control software development, including testing.

1. Control Software Development

While final development and testing of control software depends the availability of the simulation model, some work has been accomplished prior to such availability. In [Schue, 1993] an algorithm is presented for statically stable, alternating tripod gait planning and foot path planning for smooth leg motion during walking on flat terrain. Further developments in gait planning algorithms and demonstration of alternative gaits, which allow variable direction and speed but require continuous adjustment of leg liftoff and touchdown sequence, are reviewed in [Yoneda, 1993].

2. Simulation Software Development

The framework for the Aquarobot simulation model is provided in [Davidson, 1993], in which an object-oriented kinematic model is developed. Both Danevit-Hartenberg (DH) and Craig (Modified Danevit-Hartenberg or MDH) methods are presented and then compared. The fundamental difference in the two methods is in the coordinate systems used for a "link," the rigid limb component between two joints. The

DH methodology utilizes the outboard, closer to end-effector, joint as the coordinate system origin while the Craig method uses the inboard, closer to body, joint. The Craig version of Davidson's kinematic model is used in this thesis.

[Goetz, 1994] explores a variety of enhancements for the Aquarobot simulation model. A graphics model, which incorporates a surrounding operating environment (terrain), is developed to replace the original stick figure. The model includes I/O control interfaces (i.e. keyboard, joystick, spaceball) and foot/ground collision detection.

A complete, unsimplified physical dynamic simulation model of Aquarobot, including the hydrodynamic forces of its operating environment, is also being developed [McMillan, 1993]. While this simulation is not expected to run in real time, it will provide valuable data for comparison to the simplified model.

D. REUSED SOFTWARE

As mentioned above, the Aquarobot simulation presented in this thesis is based on the model described by [Davidson, 1993]. A summary of the key features of that model is provided here for quick reference.

1. Rigid Body Class

In both LISP and C++ versions of the Aquarobot model, system dynamics for six degrees of freedom, three translational and three rotational, are handled within a "rigid body class" from [Davidson, 1993]. System state variables include world coordinate position and orientation, stored in a 4x4 "body to world" coordinate transformation

matrix, called the "H-matrix", and velocities in body coordinates. Euler integrations are used for dynamic updates. Acceleration equations in body coordinates are:

$$\dot{u} = vr - wq + \frac{f_x}{m} - g\sin\theta\,; \tag{2.1}$$

$$\dot{v} = wp - ur + \frac{f_y}{m} + g\cos\theta\sin\phi\,; \tag{2.2}$$

$$\dot{w} = uq - vp + \frac{f_z}{m} + g\cos\theta\cos\phi\,; \tag{2.3}$$

$$\dot{p} = \frac{[(I_{yy} - I_{zz})qr + L]}{I_{xx}}\,; \tag{2.4}$$

$$\dot{q} = \frac{[(I_{zz} - I_{xx})rp + M]}{I_{yy}}\,; \tag{2.5}$$

$$\dot{r} = \frac{[(I_{xx} - I_{yy})pq + N]}{I_{zz}}\,; \tag{2.6}$$

where m is body mass; g is gravitational acceleration, in world coordinates; I_{xx}, I_{yy}, and I_{zz} are the moments of inertia; $f = (f_x, f_y, f_z)$ is the vector of applied forces; $T = (L, M, N)$ is the vector of applied torques; theta and phi are Euler pitch and roll angles, respectively; u, v, w are the components of translational velocity; and p, q, r are rotational rate components [Frank, 1969]. With a single exception, g, the above values are expressed in body coordinates. The dynamic update is achieved by determining incremental position and orientation changes, in body coordinates, and using those to generate an incremental motion matrix which is then post-multiplied with the body's H-matrix, and using that result to update (replace) the H-matrix. Euler integrations and equations 2.1 through 2.6 are used to update the velocity state variables [Davidson, 1993].

2. Kinematic Model

Each Aquarobot limb (leg) is made up of a series of links: the inboard end of the series being physically connected to the robot's body and the outboard end the foot. Using the Craig method, a coordinate system at the outboard end of a link is described relative to the coordinate system at the inboard end by a transformation matrix called a "T-matrix". The T-matrix depends on the physical construction of the link and, in the case of rotary links, the rotation angle of the inboard joint. The origin of each such coordinate system is the position of the joint between two such links, and the z-axis is aligned so that a joint rotation is a z-axis rotation. The entire system is kept in a hierarchical structure of "rigid bodies", with H-matrix updates done from the top down: i.e. a link's H-matrix may be updated only after the link next inboard is updated (the robot body in the case of the inboard end of each leg) [Davidson, 1993].

$$[H_i] = [H_{i-1}]\,[T_i] \qquad (2.7)$$

E. SUMMARY

The development of the Aquarobot simulation model will directly support final development and testing of Aquarobot control software. The dynamic model developed in this thesis is based on a previously developed kinematic model. Before describing the dynamic model, a more detailed description of Aquarobot and an overview of the software tools used is needed and is provided in Chapter III.

III. DESCRIPTION OF AQUAROBOT VEHICLE AND SIMULATION ENVIRONMENT

A. INTRODUCTION

This chapter provides a physical description of Aquarobot, including some details that are beyond the scope of models developed in this thesis. Special emphasis is given to the joint actuators as they are the primary feature to be modeled. In addition, the software tools used in the development are introduced.

B. PHYSICAL DESCRIPTION OF AQUAROBOT

Figure 3.1 is a photograph of Aquarobot which has a body, that is generally cylindrical in shape, and six legs, equally spaced at sixty degrees apart. Mounted on top of the body is a camera boom with three rotary joints for positioning. The boom is equipped with an ultrasonic ranging device to assist in the scaling of measurements. Within the body are two inclinometers (for attitude sensing), a gyrocompass, and a depth sensing device. Aquarobot is computer controlled from the surface via a four centimeter diameter tether cable attached to the top of the body. The cable carries control signals to the robot and returns sensor signals back to the controlling computer.

Each of the six identical insect type legs has three rotary joints, which are driven by the joint actuators described in the next section, and a freely rotating ball joint to attach the disc shaped foot. Each foot has a pressure sensitive touch sensor to provide an

indication of ground contact. Figure 3.2 illustrates the axis of rotation for each joint in an Aquarobot leg.

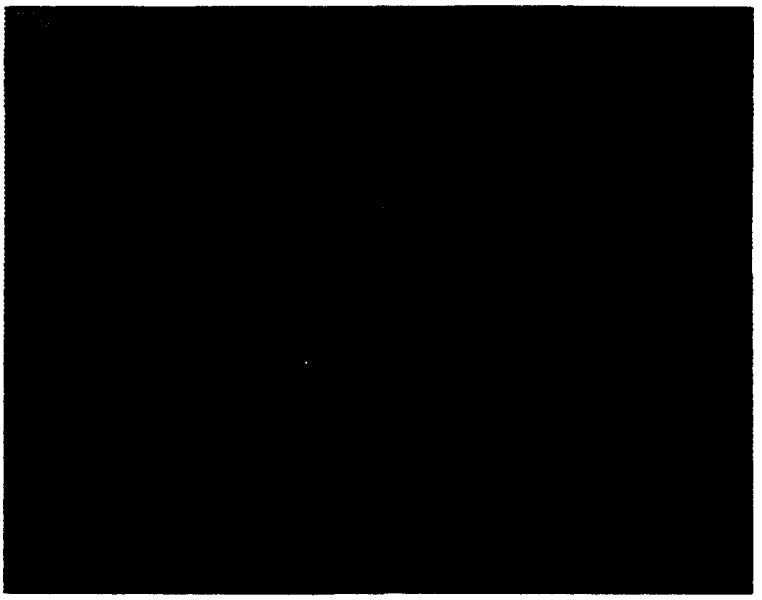


Figure 3.1
Photograph of Aquarobot

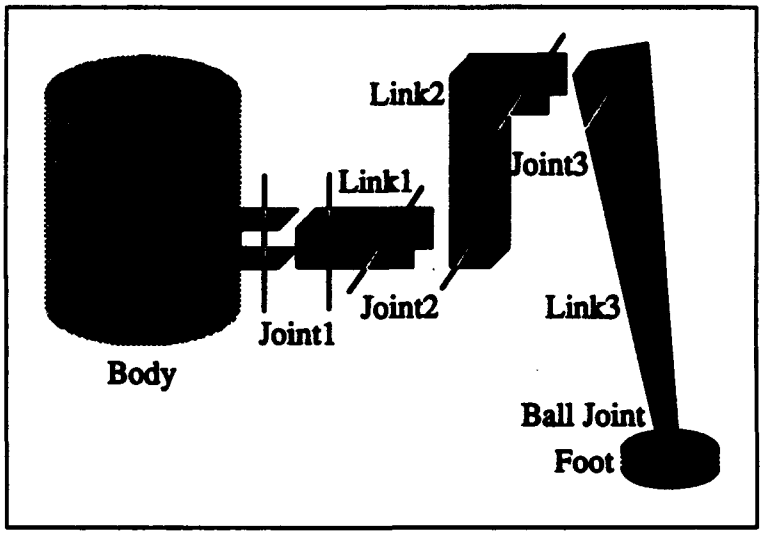


Figure 3.2
Aquarobot Leg Construction

C. AQUAROBOT JOINT ACTUATORS

Figure 3.3 is a block diagram of an Aquarobot joint actuator. Control software sends incremental rotation orders to the controller as pulses, with polarity indicating the desired direction of rotation. Motor response is fed back to the control software as pulses in the same manner. These feedback pulses are produced by a pulse generator on the motor shaft. Additional signals to and from the control software include pulse counter (PC) overflow, pulse counter clear, and driver enable.

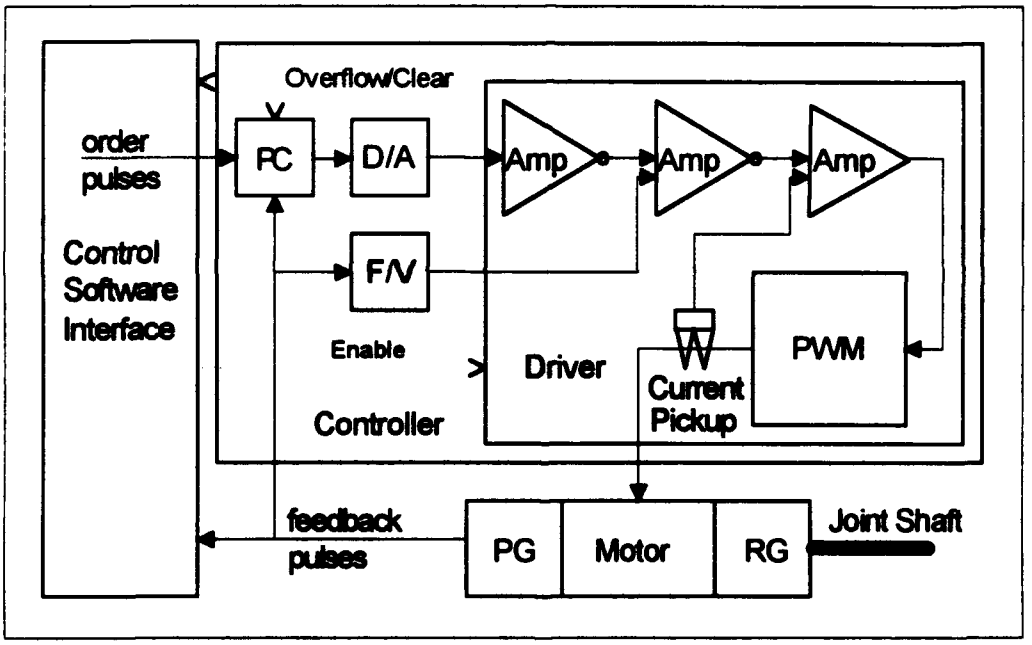


Figure 3.3
Aquarobot Joint Actuator

The difference between the actual and ordered position, the error signal of the motor, is kept in the pulse counter. Orders increment the counter, plus or minus, depending on desired direction; response pulses decrement it. One hundred pulses are required for one

motor revolution (determined by the motor shaft pulse generator output). The counter overflows if the maximum count, +/- 6144 pulses, is exceeded.

A digital to analog converter (D/A) produces a DC signal directly proportional to the current count in the pulse counter, and its output is the motor driver's primary input. That is, the larger the error, the higher the voltage applied to the motor to correct the error. A maximum count of +/-6144 is converted to +/- 10VDC, i.e.

$$DA_{out} = count * (10/6144). \tag{3.1}$$

A high gain for the error signal is desirable for fast response and for sufficient response to small errors. However, by itself, the high gain would cause severe overshoot and oscillations in the motor. Degenerative feedback, proportional to motor speed, is used to prevent, or at least minimize, this overshoot. This degenerative feedback is provided by the frequency to voltage converter (F/V) which monitors the pulse generator output. The output is 3VDC per thousand RPM's, so

$$FV_{out} = RPM * (3/1000) \tag{3.2}$$

The driver amplifies outputs from the D/A and F/V converters to produce the source voltage (V_s) for the servomotor. So far, we have

$$V_s = G_1 DA_{out} - G_2 FV_{out}, \tag{3.3}$$

where G_1 and G_2 are respective gains. However, the driver is actually more complex and includes an additional internal feedback path.

The signals form the D/A and F/V converters are amplified and fed into the pulse width modulator (PWM) with the F/V signal inserted between inverting amplifiers to

provide degeneration. A current pickup on the PWM output line provides a signal proportional the current drawn by the motor, and therefore proportional to the motor torque. Recall that for a given voltage applied to a motor, it has a limited speed due to back EMF. As the motor approaches that speed, it draws less current. This current (torque) feedback signal provides regenerative feedback in the driver during motor acceleration, thus reducing the response time. Adding the torque feedback to Equation 3.3 yields

$$V_s = G_1 DA_{out} - G_2 FV_{out} + G_3 I_a, \qquad (3.4)$$

where I_a represents the motor's armature current. Actually, this equation still neglects some complexities in the Aquarobot joint controller. Not shown in Figure 3.3 are feedback capacitors around amplifiers which further modify the equation for V_s. These effects are not modeled in the work of this thesis.

The motor is driven by a squarewave rather than a DC voltage. Figure 3.4 illustrates idealized signals. The function of the PWM is to provide a squarewave of constant amplitude, zero to +/- 75 volts, with a duty cycle proportional to the input signal, so V_s in Equation 3.4 is actually an average value. Motor response to this signal is very close to its response to a DC voltage equal to the average voltage of the squarewave. Applying Signals 1 or 2 of Figure 3.4 to the motor is thus equivalent to applying +25VDC or -25VDC, respectively. This methodology is used primarily for its efficiency advantage over a DC linear amplifier [Truxal, 1958].

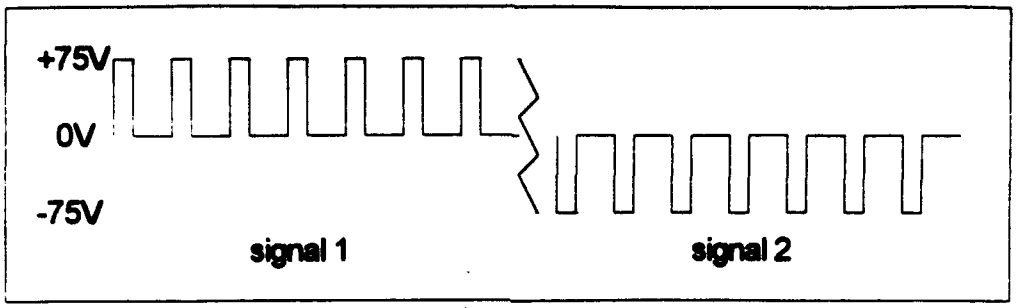


Figure 3.4
PWM Outputs For 33 % Duty Cycles

Attached to each servomotor is a harmonic reduction gear, built together as a single unit. The model used for joint one in each leg, RA-20, has a reduction ratio of 161:1, while that used for joints two and three, RH-25, has a 160:1 ratio. In addition, joints two and three have a bevel gears cascaded with the harmonic gears with 3:1 and 2:1 ratios, respectively. Figure 3.5 summarizes the total reduction ratios and gives the control pulses required for a single joint shaft revolution.

Joint	1	2	3
Harmonic Gear	161:1	160:1	160:1
Bevel Gear	NA	3:1	2:1
Total Gear Ratio	161:1	480:1	320:1
Pulses / Revolution	16,100	48,000	32,000

Figure 3.5
Total Reduction Gear Ratios and
Control Pulse Requirement per Joint Revolution

D. SOFTWARE TOOLS

C++ is the programming language selected for modeling Aquarobot. This choice was based on both hardware and software considerations. IRIS Performer, a C/C++ visual simulation toolkit, provides rendering that is fast enough to display a real time dynamic simulation [Goetz, 1994]. Common LISP was used for rapid prototyping and testing during the early stages of development.

1. C++

In a long term project with many contributors, object oriented design provides a high level of abstraction which promotes rapid system level comprehension by new team members as well as loosely coupled, easily maintained source code. Not only does C++ support the object oriented design paradigm, but also, it is based on, and includes as a subset, the highly efficient, structured language, C [Wiener, 1988].

The simulation platform, both here and at PHRI, is an IRIS Reality Engine. The C/C++ compilers delivered with the IRIS systems are very efficient, due to direct access to the low level hardware, and are intended to be used as native development languages.

Popularity and widespread use make C/C++ source code portable, and while portability is not a key issue for the Aquarobot simulation, it is very much so for the Aquarobot control software which is likely to have to survive hardware upgrades.

2. Performer

IRIS Performer is used primarily for its performance as a rendering tool. Performer is a hardware oriented, C/C++ graphics tool kit designed to operate on Silicon Graphics products. Its routines take full advantage of the "known hardware" platform to allow much higher performance than routines written for generic hardware. Also, some precision is traded for speed as real numbers in Performer data structures are single precision "floats" rather than "doubles". Upon initialization, Performer detects hardware capability and automatically sets up a multiprocessing environment with shared memory when running on a multiprocessor machine. The default configuration, which the user may override, is separate processors assigned for (1) user application, (2) graphics database culling, and (3) drawing the culled database to a graphics window [SGI, 1992].

Performer's graphics database is stored in a hierarchical data structure, a tree. The structure is culled and rendered by doing a pre-order traversal with child nodes inheriting the accumulated environments (transformations) of all ancestors traversed in the current path. These accumulated environments are kept on a stack and are popped off when traversing back up the tree. The basic nodes are Scene (the root), Static Coordinate System (SCS), Dynamic Coordinate System (DCS) (variable for motion where an SCS is fixed), Group (a container for children requiring a common transformation), and Geode (a container for polygons to be rendered). Geodes must be leaves but may have more than one parent. This reduces the memory requirements when a group of polygons is to be rendered more than once in a frame (two or more identical objects) [SGI, 1992].

Robotics and mechanics users must be aware that graphics standards are used in matrix construction and operations. The graphics matrix is the transpose of the standard robotics matrix [Fu, 1987], $GM_{ij} = RM_{ji}$. This is only important when using the pfMatrix data type and operations outside those that directly support the graphics database, or when directly manipulating individual elements, rows, and / or columns. Since $MxN = (N^{T}xM^{T})^{T}$, the order of matrix multiplications may also have to be reversed. Lastly, X and Y axis rotations are as expected; however in Performer, pitch refers to an X, rather than Y, axis rotation, and likewise, roll refers to a Y axis rotations [SGI, 1992].

Synchronization for a real-time application may be achieved by setting Performer's "desired frame rate" and then using the pfSync function call. The pfSync function will put all processes to sleep between each frame to force the desired frame rate. If the desired frame rate is faster than can be achieved, pfSync will have no effect, and the application will free run at its fastest speed. In the Aquarobot simulation, the frame rate is set to twenty frames per second and then a fixed delta-time of one twentieth of a second is used for dynamic updates. Reading the internal clock for delta-time would have been equivalent providing the application is capable of running at the desired frame rate. The fixed delta-time ensures control over data points [SGI, 1992].

3. LISP

The problem with "rapid, experimental" prototyping in C/C++, as well as other compiled languages, is the difficulty with testing and verification. Specifically, a test shell must be written, compiled and run to thoroughly test a block of source code. If the test

results in the detection of errors, debugging tools are available, but the source code must be recompiled to include them. Other methods, such as insertion of additional lines of code, are also available but also require multiple compilations. In any case, logic errors are still difficult to find when they exist in large blocks of code, especially when they only apply to specific inputs. LISP, on the other hand, is an ideal language for experimental work. LISP is an interpreted, functional language which gives the developer the ability to call any defined function directly from the command prompt. The function's return value is immediately displayed for easy comparison to expected values; test routines are not required. Since functions may be nested, these test calls may include any desired level of abstraction. Variables are also directly accessible in LISP and may be inspected at any time. The basic structure of a LISP program is an on-line database of definitions: constants, functions, and symbols (pointers to variables) [Koschmann, 1990].

Weak typing in LISP allows additional flexibility. Typing is done dynamically at run time. For example, a single definition of the function "minimum" may be used for integers, reals, or any argument type for which the operator "<" is defined. Variables, arguments, and return values may also be lists, allowing the developer to call a function with a comprehensive set of test inputs [Koschmann, 1990].

The Common LISP Object System, CLOS, provides full support for object oriented programming. In this thesis, a prototype design using CLOS is first developed to implement system design decisions. Subsequent translation to C++ is then accomplished without abstract structure modifications.

E. SUMMARY

This chapter physically describes Aquarobot with the intention of providing sufficient orientation prior to the model presentations. Several Aquarobot features are not included in the dynamic simulation model presented later in Chapter V. The kinematic model includes the body, legs and feet but not the camera boom. Joint position, foot contact, and azimuth information are available; however, attitude and depth are not. The tether cable and hydrodynamic forces, currents and viscosity, are also neglected.

Also in preparation for the model presentations, the software tools utilized, along with the reasons for their selections, are introduced. Object-oriented system design and the need for a high performance made C++ and IRIS Performer ideal software tools for the Aquarobot simulation model. CLOS, with its capability for rapid testing of complex functions, was used for prototyping and model verification.

The following chapters present dynamic simulation models of an Aquarobot joint actuator and Aquarobot itself.

IV. JOINT ACTUATOR MODEL

A. INTRODUCTION

The purpose of this chapter is to present a simulation model for a single Aquarobot joint actuator. It begins with a review of the basic mathematics required to model servomotors and reduction gears and then develops the joint actuator model.

B. BASIC DC MOTOR

A motor is a device used to convert electrical energy into mechanical energy. The force, F, on a current, i, carrying conductor of length l in a uniform magnetic field, B, is given by [Halliday, 1981]

$$F = \int idl \times B. \tag{4.1}$$

If the conductor is fixed on a shaft, parallel to the shaft, at radius r, the resulting torque, *tau*, is given by [Halliday, 1981]

$$\tau = r \times F. \tag{4.2}$$

The motor's armature includes a set of such conductors and is usually constructed so as to be symmetrical with respect to the axis of rotation; therefore, the total conductor length and relative position (to the magnetic field) is independent of the armature's angular position. Furthermore, if a constant magnetic field (i.e. permanent magnet or constant

current electromagnet) is used, then the force and resulting torque become directly proportional to the armature current, simplifying the above equations to

$$F = iC, \tag{4.3}$$

and

$$\tau_d = iK_t, \tag{4.4}$$

where tau_d is developed torque, and torque constant K_t is a characteristic of the motor.

The armature current depends on applied voltage, armature resistance, and armature angular velocity, *omega*. The velocity dependency is due to the voltage induced in the conductors as they move through the field (Faraday's Law). Since this induced voltage is of such polarity that it causes a decelerating torque (Lenz's Law), it is called Back or CounterEMF, V_b [Halliday, 1981] [McPherson, 1981]

$$V_b = K_b\omega, \tag{4.5}$$

where K_b is the motor's back EMF constant.

Armature inductance is usually negligible for high quality servomotors, so using Ohm's Law, the armature current, I_a, is given by [Halliday, 1981] [McPherson, 1981]

$$I_a = \frac{V_a - V_b}{R_a}, \tag{4.6}$$

where V_a is the voltage applied to the armature and R_a is armature resistance.

Combining Equations 4.4 through 4.6, the motor's developed torque is

$$\tau_d = I_a K_t = \frac{K_t(V_a - K_b\omega)}{R_a}. \tag{4.7}$$

Given no external torques and ignoring losses for now, for any V_a there is an associated maximum speed, when $V_a = K_b$ *omega*, that results in $tau_d = 0$ and therefore no further

acceleration. Still ignoring losses, motor acceleration, *omega-dot*, is given by the standard rotational dynamics equation [Halliday, 1981]

$$\dot{\omega} = \frac{torque}{inertia} = \frac{\tau_d + \tau_x}{J_m + J_x}, \tag{4.8}$$

where J_m is internal motor inertia, and tau_x and J_x are external torque and inertia, respectively. There are several sources for losses in a motor, but as long as the motor is operated within its prescribed limits, most of them may be consolidated into two groups: constant, F_c, and directly proportional to velocity, F_v *omega*. Some examples are friction, copper (i^2R), and windage [McPherson, 1981]. External loads may also include constant and velocity dependent losses. The actual loss is state dependent and requires some special handling:

(1) shaft turning (omega <> 0) : loss opposes omega

$$|loss| = F_c + F_v|\omega|; \tag{4.9}$$

(2) shaft at rest (omega = 0) : loss opposes torque

(a) torque sufficient to overcome friction (|torque| > Fc)

$$loss = F_c; \tag{4.10}$$

(b) torque insufficient to overcome friction (requires *torque* - *loss* = 0)

$$loss = torque. \tag{4.11}$$

State 2a may be handled by the method used for state 1, but since it is already detected by the test for state 2b, there is no reason to perform the arithmetic. Incorporating losses into Equation 4.8 yields

$$\dot{\omega} = \frac{\tau_d + \tau_x - Losses}{J_m + J_x}. \tag{4.12}$$

The only other significant loss not fitting into one of the groups above is the voltage drop across the motor brushes, brush drop loss V_{bd}. The voltage applied to the motor armature, V_a, is the motor source voltage, V_s minus this brush drop loss [McPherson, 1981].

$$V_a = V_s - V_{bd} \tag{4.13}$$

Combining Equations 4.7, 4.12 and 4.13, motor response depends on construction, state, applied voltage, and load (external torque and inertia).

$$\dot{\omega} = \frac{\left[\left(K_t \frac{V_s - V_{bd} - (K_b \omega)}{R_a}\right) + T_x - Losses\right]}{J_m + J_x} \tag{4.14}$$

As a final note, motors are given voltage and load ratings. These are determined by motor construction and are intended to keep armature current within safe operating limits [McPherson, 1981].

C. REDUCTION GEAR

A reduction gear is a mechanical coupling device that provides a mechanical advantage allowing a higher speed, lower torque source to drive a lower speed, heavier load. This enables both source and load to operate at or near their optimal, most efficient speeds, even though those speeds are not the same. The gear ratio, n, is the ratio of the input and output shaft angular displacements, velocities and accelerations, *theta*, *omega* and *omega-dot*, respectively [Chen, 1993].

$$\theta_{out} = \frac{\theta_m}{n}, \tag{4.15}$$

$$\omega_{out} = \frac{\omega_m}{n}, \tag{4.16}$$

$$\dot{\omega}_{out} = \frac{\dot{\omega}_m}{n}. \tag{4.17}$$

While the output shaft speed is reduced by a factor of n, the torque is increased by a factor of n [Chen, 1993].

$$\tau_{out} = n\,\tau_{in} \tag{4.18}$$

Recalling and rearranging Equation 4.8 and applying it to the output shaft

$$J_{out} = \frac{\tau_{out}}{\dot{\omega}_{out}}. \tag{4.19}$$

Substituting for $omega\text{-}dot_{out}$ and tau_{out}, using Equations 4.17 and 4.18

$$J_{out} = \frac{n\,\tau_{in}}{\left(\dot{\omega}_{in} / n\right)} \tag{4.20}$$

$$= n^2 \frac{\tau_{in}}{\dot{\omega}_{in}} \tag{4.21}$$

$$= n^2\, J_{in} \tag{4.22}$$

This gives a coupling factor for inertia of n^2, i.e.

$$\frac{J_{out}}{J_m} = n^2. \tag{4.23}$$

Combining a motor and a reduction gear as a drive train for some load, Equations 4.12, 4.18 and 4.22, yields:

$$\dot{\omega}_m = \frac{\tau_d + (\tau_x/n) - Losses}{J_m + \left(J_x/n^2\right)}, \text{ or} \tag{4.24}$$

or

$$\dot{\omega}_x = \frac{(n\,\tau_d) + \tau_x - (n\,Losses)}{\left(J_m\, n^2\right) + J_x}, \tag{4.25}$$

where m is the input (motor) side of the reduction gear, and x is the output side.

D. JOINT ACTUATOR SIMULATION

Figure 4.1 is a block diagram of the Joint Actuator Simulation Model. Refer to Figure 3.3 for a comparison to the actual Aquarobot Joint Actuator. Since the model was in essence a "bench top" device, physical limitations, such as pulse counter overflow and motion limits, were not detected or enforced. Also, input pulses were replaced by a change in desired angular position of the joint shaft in revolutions, delta-theta, and the motor driver's rectangular wave output is simplified to its average value. Loading and operating instructions and a complete source code listing for this model may be found in Appendix A.

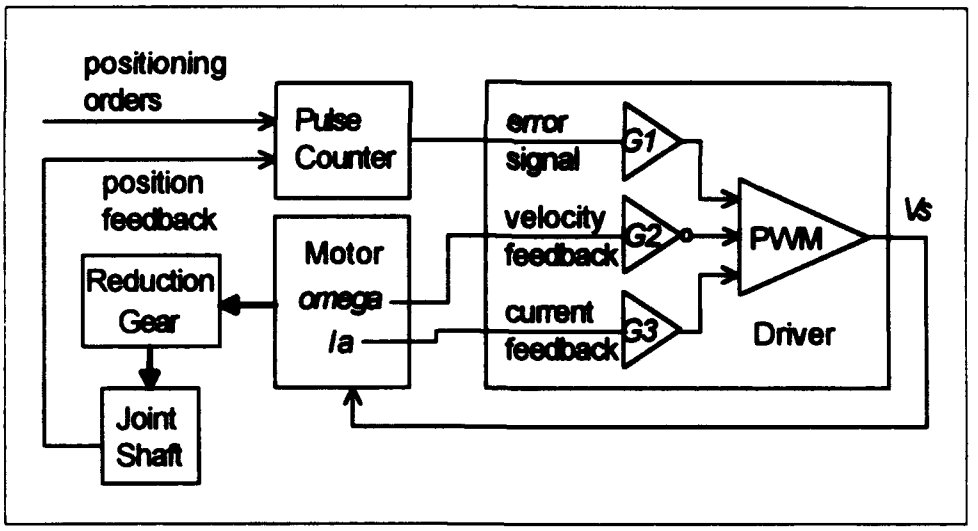


Figure 4.1
Joint Actuator Simulation Model

The D/A converter is not necessary in the simulation model as the Pulse Counter stores the difference between the current and ordered shaft positions rather than a discrete pulse count. Motor velocity and armature current, slots values in the motor class, are

directly accessed to provide those inputs to the driver. The cascaded amplifiers in the driver are separated and then summed in order to simplify gain adjustments. Positional feedback is taken after the reduction gear to match the implementation of the positioning orders input. This is in contrast to the physical device in which feedback pulses come directly from the motor itself.

1. Base Classes

Several base classes are used in the joint actuator model. Refer to Appendix A for a source code listing. The first is the "diff-counter" class used to model the pulse counter. A single slot, "current-count", which is initialized to zero, holds the cumulative sum of all delta theta orders and feedback. Its single method, "diff-signal", updates and returns the current count.

$$count = count + order - feedbac \tag{4.26}$$

The "amplifier-clipper" class has three slots: "amplification-factor" (or gain), "max-value" and "min-value". The "amplify" method simply multiplies the argument by the gain and returns the product, clipped of course, to the maximum or minimum value if the product exceeds those prescribed limits.

The "driver-class", a sub-class of amplifier-clipper, adds three gain slots for independent amplification of the three inputs before summing them in the final amplifier-clipper stage. The current feedback is not internal but relies on a saved slot value in the motor-class.

The "shaft" class is used as a superclass for the motor class, as slots for reduction gear input and output shafts, and for the actual joint shaft. Slots angular-position (theta), angular-velocity (omega), inertia, coulomb-friction-constant (Fc or constant loss component), viscous-friction-constant (Fv or velocity dependent loss component), and time-stamp are defined. Methods are defined to provide capability to set theta and omega to some position and speed, reset theta and omega to zero, and couple (transfer) theta and omega to another shaft.

The "motor" class inherits from the shaft class and defines additional slots: torque-constant (K_t), back-emf-constant (K_b), armature-resistance (R_a), armature-current (i_a), and brush drop parameters (*max-brush-drop* and *half-brush-drop-source-value)*. Methods "developed-torque" and "omega-dot" are direct implementations of Equations 4.7 and 4.14, respectively. Brush drop was approximated by using an exponential form that approaches *max-brush-drop* as the source voltage increases [McPherson, 1981].

$$V_{bd} = V_{bd\max}\,(1 - 0.5^{\,|V_s|/V_{b\text{-half-value}}}) \qquad (4.27)$$

Method "run-motor" gets the elapsed time (*dt*), calculates *omega-dot*, then updates the motor state using Euler integrations.

The "reduction-gear" class has slots "gear-ratio" (*n*), "in-shaft" and "out-shaft", the last two being instances of class shaft. Methods are provided to multiply or divide an argument by n (n squared if the argument is inertia). The method "rg-connect" is provided to replace the shaft couple method for internal coupling and reduces the coupled theta and omega by the gear ratio.

2. Joint Class

The "joint" class actually represents a joint-actuator. It includes the entire system: motor controller, servo-motor (prime-mover), and reduction-gear. An additional shaft slot, "load-shaft", is actually only included to allow a convenient method to store the previous theta value in order to determine the feedback delta theta. The output shaft of the reduction gear actually holds the remaining load shaft slot values. The motor could likewise have been the reduction gear input shaft; however, they were kept separate for clarity.

Methods are provided to sum system parameters external to the motor (i.e. load-inertia, ...) so they may be passed into calls to run-motor. "Feedback" returns the difference between the output shaft of the reduction gear and the load shaft. "Increment-joint" calls "run-motor" then couples all the shafts, being careful to save the old reduction gear output shaft position first for use in next call to feedback. "Step-input-to-joint" provides the facility to send delta theta orders to the pulse counter, and "reset-joint" reinitializes the system.

Loading "joint.instance.cl" creates the instance of a joint used for model testing. The functions provided allow various orders to be sent to the joint and then displays system response to those orders. "Move-joint-mult" orders multiple repetitions of the same "delta-theta" order, each order being initiated upon completion of the previous order. "Move-joint-list" is similar in execution, but takes a list (sequence) of delta-theta's rather than repeating the same one. "Run-joint" orders a continuous sequence of small

delta-thetas, determined by argument "speed" and system elapsed time, required to achieve the ordered RPM speed. The remaining functions are intended to be internal calls. They are "clear-and-reset" (called by each of the previous three for initialization), "move-joint-list-2" (recursive call for move-joint-list), and "move-joint" (called by move-joint-mult and move-joint-list-2). "Move-joint" is the workhorse and makes repeated calls to method "increment-joint" until the ordered delta-theta has been achieved (the pulse counter's current count approaches zero). In contrast, "run-joint" makes repeated calls to "increment-joint" until the ordered speed is achieved.

3. Additional Supporting Code

Loading "Window.instance.cl" creates a display window with a gradicule. A call to "display-state" reads the appropriate state slots and draws the set of data points for the current time. Additional methods are provided for internal calls and for reinitialization (i.e. clearing or resetting the window). The end result is a display of system state vs. time for program output. Finally, the file "joint-loader" is provided as a convenience, and loading it ensures loading of the source code files in the correct sequence (dependencies are observed). After loading the source code, it then makes calls that test run various features of the system.

E. SUMMARY

This chapter provides a review of the basic mathematics required to model servomotors and reduction gears and then describes the simulation of an Aquarobot joint

actuator. The model is simplified in that exception handling control signals are ignored and the pulse counter input is desired delta-theta for the joint shaft, but functionally, it is equivalent. In the next chapter, the Aquarobot model is presented; however, joint actuators are not yet included but instead are functionally represented by springs and dampers.

V. SPRING AND DAMPER MODEL

A. INTRODUCTION

A modified dynamic simulation model for Aquarobot is developed in this chapter. It has springs and dampers in place of servo motors to provide joint torques. The springs and dampers are intended to eventually be replaced by a joint actuator model such as that presented in the previous chapter. The model can be tested by giving the Aquarobot an initial position and orientation and then allowing it to drop, observing the response as the feet contact the ground and the legs provide support.

Since the purpose of the model is to provide an approximate dynamic simulation, capable of running in real time, several simplifications have been made:

(1) the legs are considered to be massless;

(2) the ground-foot friction is infinite (no slippage);

(3) the center of mass is assumed to be at the geometric
center of the inboard joints of the legs; and

(4) body inertia is that of a solid homogeneous cylinder.

In addition, joint stops (physical limits) in the kinematic model are disabled and collisions, other than feet contacting the ground, have been ignored.

B. INVERSE KINEMATICS

The Aquarobot leg kinematic model allows determination of foot position, given the joint angles. The inverse kinematic problem, on the other hand, is to determine the joint angles, given the foot position. Using the appropriate coordinate systems simplifies inverse kinematics. Figure 5.1 illustrates the coordinate system used to calculate joint1's angle, theta 1: the joint is the origin, the Z-axis is down (parallel to body Z-axis) and the X-axis is radially outward from the body. Theta1 is measured as a right handed Z-axis rotation using the X-axis as a zero reference.

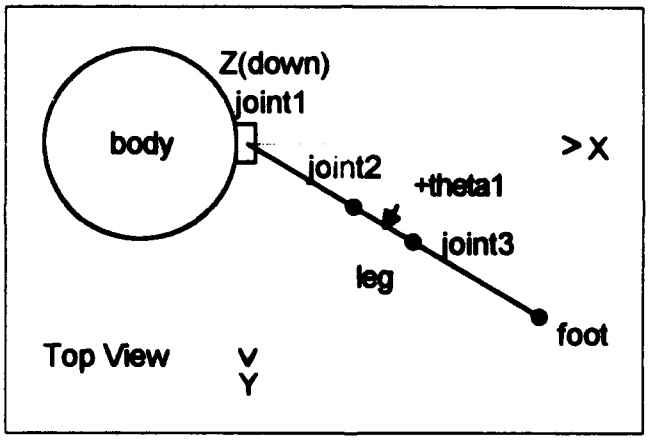


Figure 5.1
Top View of Joint1 Coordinate System Used to Calculate Theta1.

Given the foot position in this coordinate system, theta1 is easily calculated:

$$\theta_1 = \arctan\left(\frac{y_{foot}}{x_{foot}}\right). \qquad (5.1)$$

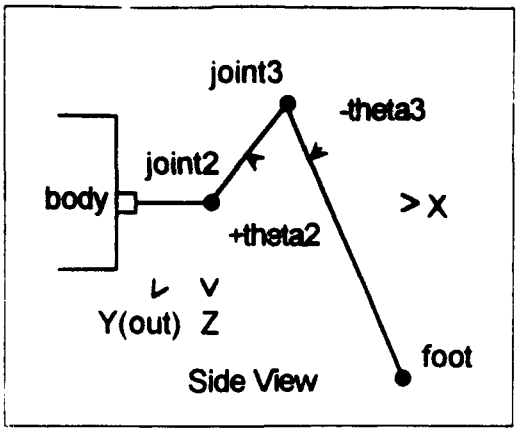


Figure 5.2
Side View of Coordinate System Used to Calculate Theta2 and Theta3.

Figure 5.2 illustrates the coordinate system used for calculating theta2 and theta3. Joint2 serves as the origin, the X-axis is defined as the direction directly away from joint1, and the Z-axis is again down. Using this coordinate system reduces the problem to two dimensions as the y-component of the foot position is now zero. Figure 5.3 and Equation 5.2 illustrate the *law of cosines* [Oakley, 1971] which is used to determine theta2 and theta3.

$$a^2 = b^2 + c^2 - 2bc\cos \qquad (5.2)$$

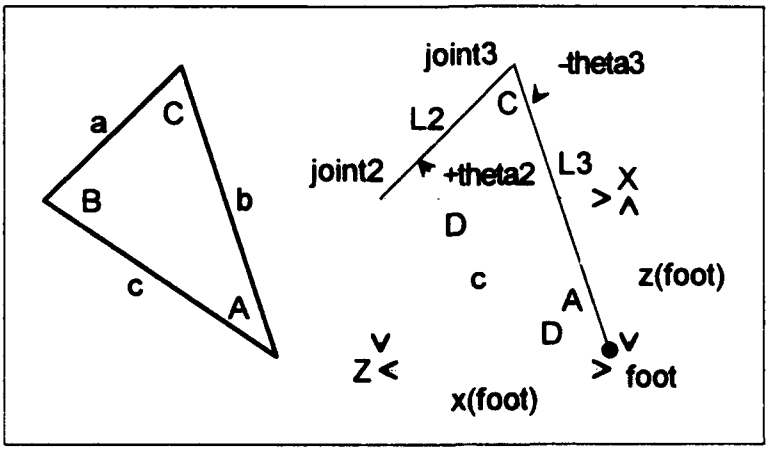


Figure 5.3
Applying Law of Cosines

Solving Equation 5.2 for angle A:

$$A = \arccos\left[\frac{b^2+c^2-a^2}{2bc}\right]. \tag{5.3}$$

Angles B and C are determined in the same manner:

$$B = \arccos\left[\frac{a^2+c^2-b^2}{2a}\right], \tag{5.4}$$

$$C = \arccos\left[\frac{a^2+b^2-c^2}{2ab}\right]. \tag{5.5}$$

Referring back to Figure 5.3, sides a and b correspond directly to the lengths of link2 and link3, respectively, and side c may be calculated using Pythagorean's theorem:

$$c = \sqrt{x_{foot}^2 + z_{foot}^2}. \tag{5.6}$$

Angle D may be determined by several trigonomic functions; arcsin is used here:

$$D = \arcsin\left[\frac{z_{foot}}{c}\right]. \tag{5.7}$$

Theta2 and theta3 are measured using the sign convention shown. Note that theta3 is the negative of C's compliment:

$$\theta_2 = B - D, \tag{5.8}$$

and

$$\theta_3 = C - \pi. \tag{5.9}$$

Combining Equations, (5.4, 5.7 and 5.8) and (5.5 and 5.9), and substituting L2 and L3 for a and b results in

$$\theta_2 = \arccos\left[\frac{L_2^2 + c^2 - L_3^2}{2 L_2 c}\right] - \arcsin\left[\frac{z_{foot}}{c}\right], \text{ and} \tag{5.10}$$

$$\theta_3 = \arccos\left[\frac{L_2^2 + L_3^2 - c^2}{2 L_2 L_3}\right] - \pi. \tag{5.11}$$

C. JACOBIAN MATRIX

The Jacobian Matrix, $J_r(q)$ or simply J_r, of vector r with respect to vector q is defined:

$$J_r(q) = \left[\frac{\partial r_i}{\partial q_j}\right], \tag{5.12}$$

and is used to express the relation between an end effector velocity and the joint velocities of a manipulator [Yoshikawa, 1990]. In the case of Aquarobot, the foot velocity with respect to leg joint velocities is given by:

$$\begin{bmatrix} \dot{x}_{foot} \\ \dot{y}_{foot} \\ \dot{z}_{foot} \end{bmatrix} = [J_{foot}]_{3\times 3} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix}. \tag{5.13}$$

By rearranging Equation 5.13, the *inverse* Jacobian and the foot velocity may be used to determine the joint velocities:

$$\begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix} = [J_{foot}]^{-1} \begin{bmatrix} \dot{x}_{foot} \\ \dot{y}_{foot} \\ \dot{z}_{foot} \end{bmatrix}. \tag{5.14}$$

The Jacobian will also be used in the next section to determine ground reaction forces.

The derivation of the Jacobian matrix for an Aquarobot leg is straightforward. The foot position of an Aquarobot leg is kinematically described as a function of the joint angles in [Schue, 1993]:

$$x = L_0\cos\theta_0 + L_1\cos\theta_{01} + L_2\cos\theta_{01}\cos\theta_2 + L_3\cos\theta_{01}\cos\theta_2, \tag{5.15}$$

$$y = L_0\sin\theta_0 + L_1\sin\theta_{01} + L_2\sin\theta_{01}\cos\theta_2 + L_3\sin\theta_{01}\cos\theta_{23}, \tag{5.16}$$

$$z = -L_2\sin\theta_2 - L_3\sin\theta_{23}, \tag{5.17}$$

where L_i is $link_i$ length, $theta_i$ is $joint_i$ angle, $theta_{ij}$ is the sum of $theta_i$ and $theta_j$, and $link_0$ represents the constant pseudo-link from center of body to $joint_1$ and has a joint angle equal to the "leg attachment angle". Differentiating Equations 5.15 through 5.17 gives:

$$\dot{x} = -L_1\left(\sin\theta_{01}\,\dot{\theta}_1\right) - L_2\left(\sin\theta_{01}\cos\theta_2\,\dot{\theta}_1 + \cos\theta_{01}\sin\theta_2\,\dot{\theta}_2\right)$$

$$-L_3\left(\sin\theta_{01}\cos\theta_{23}\,\dot{\theta}_1 + \cos\theta_{01}\sin\theta_{23}\left(\dot{\theta}_2 + \dot{\theta}_3\right)\right), \qquad (5.18)$$

$$\dot{y} = -L_1\left(\cos\theta_{01}\,\dot{\theta}_1\right) - L_2\left(\cos\theta_{01}\cos\theta_2\,\dot{\theta}_1 + \sin\theta_{01}\sin\theta_2\,\dot{\theta}_2\right)$$

$$-L_3\left(\cos\theta_{01}\cos\theta_{23}\,\dot{\theta}_1 + \sin\theta_{01}\sin\theta_{23}\left(\dot{\theta}_2 + \dot{\theta}_3\right)\right), \qquad (5.19)$$

$$\dot{z} = -L_2\left(\cos\theta_2\,\dot{\theta}_2\right) - L_3\left(\cos\theta_{23}\left(\dot{\theta}_2 + \dot{\theta}_3\right)\right). \qquad (5.20)$$

Regrouping and translating Equations 5.18 through 5.20 into the form of Equation 5.13, the Jacobian Matrix is given by

$$J = \begin{bmatrix} -(L_1 + L_2\cos\theta_2 + L_3\cos\theta_{23})\sin\theta_{01} & -(L_2\sin\theta_2 - L_3\sin\theta_{23})\cos\theta_{01} & -L_3\cos\theta_{01}\sin\theta_{23} \\ (L_1 + L_2\cos\theta_2 + L_3\cos\theta_{23})\cos\theta_{01} & -(L_2\sin\theta_2 - L_3\sin\theta_{23})\sin\theta_{01} & -L_3\sin\theta_{01}\sin\theta_{23} \\ 0 & -L_2\cos\theta_2 - L_3\cos\theta_{23} & -L_3\cos\theta_{23} \end{bmatrix}$$

(5.21)

D. FORCES ON AQUAROBOT

Assuming homogeneous cylindrical distribution of body mass and massless legs reduces the complexity of the forces and torques on Aquarobot. As Figure 5.4 illustrates, the resulting summations of these forces and torques may be expressed

$$\vec{f}_{Aquarobot} = mg + \sum_{leg=1}^{6} \vec{f}_{leg}, \qquad (5.22)$$

and

$$\vec{\tau}_{Aquarobot} = \sum_{leg=1}^{6} \vec{r}_{leg} \times \vec{f}_{leg}, \tag{5.23}$$

where f_{leg} is ground reaction force, and r_{leg} is the moment arm, from the body's center of mass to the foot, on which that force is exerted.

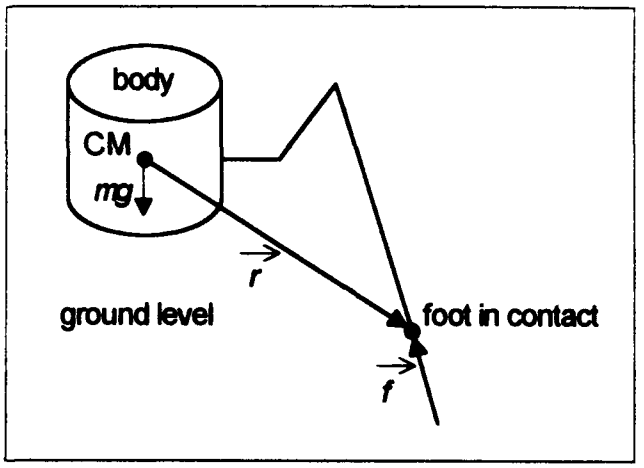


Figure 5.4
Forces on Aquarobot

Given an initial known state (position, orientation and velocity), Aquarobot's motion may be completely described by application of Newton's second law. The kinematic solution for r_{leg} is already available [Davidson, 1993], so all that remains is determination of f_{leg}.

The torques at the joints in a manipulator are related to the force exerted by the end effector by the transpose of the Jacobian [Yoshikawa, 1990]:

$$T = \begin{bmatrix} \tau_{\theta_1} \\ \tau_{\theta_2} \\ \tau_{\theta_3} \end{bmatrix} = [J]^T \vec{f}. \tag{5.24}$$

where f is the force the foot exerts on the ground, equal magnitude but opposite direction of ground reaction force. To avoid confusion, f will thus refer to the ground reaction force, the force the ground exerts on the foot, as it is the force in which we are interested; therefore Equation 5.24 must have a sign change:

$$T = [J]^T\left(-\vec{f}\right). \tag{5.25}$$

Solving Equation 5.25 for f yields:

$$\vec{f} = -[J^T]^{-1}T. \tag{5.26}$$

E. SPRING AND DAMPER JOINT TORQUES

In the spring and damper model, joint torque is the sum of spring restoring torque and damping torque:

$$\tau_{joint} = \underbrace{(-k_s(\theta - \theta_0))}_{\tau_{spring}} + \underbrace{\left(-k_d\dot{\theta}\right)}_{\tau_{damp}}, \tag{5.27}$$

where k_s and k_d are spring and damping constants, and theta0 is the ordered position. For the remainder of this chapter, theta0 will be taken as the rest position or zero torque reference. For convenience, the symbols on the right side of Equation 5.27 will be used to refer to their vector forms and will represent all three joints in a leg. Replacing the left side with T, and assuming the same spring and damping constants for each joint,

$$T = -k_s\left(\begin{bmatrix}\theta_1\\ \theta_2\\ \theta_3\end{bmatrix} - \begin{bmatrix}\theta_{1_0}\\ \theta_{2_0}\\ \theta_{3_0}\end{bmatrix}\right) - k_d\begin{bmatrix}\dot{\theta}_1\\ \dot{\theta}_2\\ \dot{\theta}_3\end{bmatrix}. \tag{5.28}$$

The joint angles are determined using the inverse kinematic method described above and are then available for the Jacobian matrix. For any foot in contact with the ground, its velocity, relative to the body, is simply the negative of the sum of the body's translational velocity and the cross product of the body's rotational velocity with the foot's position vector. Combining Equations 5.14 and 5.27, and substituting for foot velocity gives us:

$$T = -k_s\,(\theta - \theta_0) + k_d\,J^{-1}\,[v_{body} + (\omega_{body} \times r_{foot})], \tag{5.29}$$

which is now substituted into Equation 5.26 to give us the ground reaction force:

$$\vec{f} = [J^T]^{-1}[k_s(\theta - \theta_0) - k_d J^{-1}[v_{body} + (\omega_{body} \times r_{foot})]\,. \tag{5.30}$$

F. LISP PROTOTYPE

The kinematic model for the Aquarobot simulation was borrowed from [Davidson, 1993] with the only code modification being the conversion to Modified Danevit-Hartenberg (MDH) coordinate systems to match the C++ version. The complete source code listing with loading and operating instructions may be found in Appendix C. The additions required for dynamic operation of the model are best presented by the following walk-through of a dynamic update. Figure 5.5 provides a flowchart for reference.

A dynamic update of Aquarobot is achieved by calling method "update-aquarobot". The body's acceleration, velocity and position are all updated by calls to previously defined methods in the rigid-body class. After these are completed, and the body is repositioned, the legs, forces and torques are then updated by calling "update-forces-and-torques".

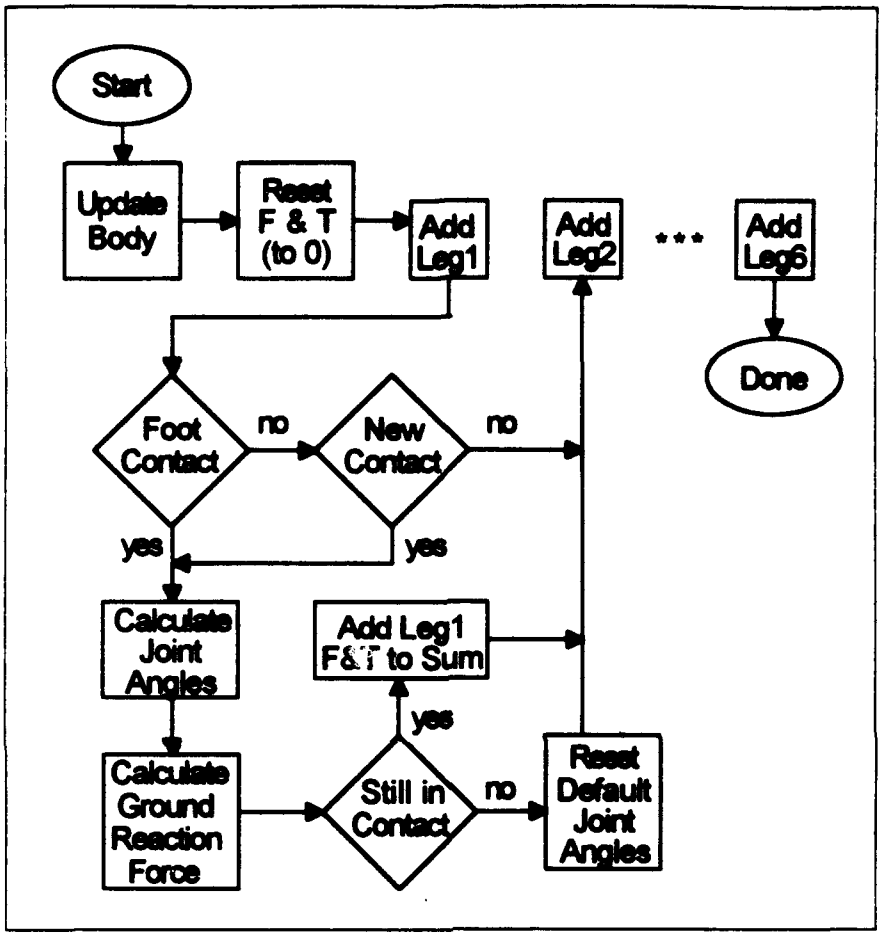


Figure 5.5
Flowchart for Dynamic Update of Aquarobot.

Gravity is handled separately in the "update-acceleration" method; therefore, only the forces and torques due to foot contact with the ground need be considered. Update-forces-and-torques resets the body's forces-and-torques vector, a rigid-body slot, to zeros, then calls "add-leg-forces-and-torques" for each leg to generate an updated cumulative value.

Add-leg-forces-and-torques tests for "foot-contact", a Boolean type slot, or "new-contact", a function that sets foot-contact to true and the foot's world z coordinate to zero (ground level) if ground penetration is detected (z coordinate greater than zero). If there is no foot contact, nothing happens: joint angles remain set to their default values and the body's cumulative forces-and-torques value is left unchanged. If however, there is contact, the inverse kinematics routine is called, the ground reaction force is calculated using Equation 5.30, and the joint angles are updated. Before updating the cumulative forces-and-torques, loss of contact must be detected. This is done by testing the world z component of the calculated ground reaction force in a call to "still-in-contact". If the force is such that it is pulling the robot down rather than supporting it, then foot-contact is set to false, the joint angles are reset to their default values, and again, no contribution to forces-and-torques. If the foot is determined to be still-in-contact, world z component of the force less than zero (pushing up), then method "add-forces-and-torques-to-body" is called which adds f and $r \times f$ to the cumulative value of forces-and-torques as in Equations 5.22 and 5.23. After cycling through all the legs, Aquarobot is completely updated and ready for another cycle.

This dynamic update cycle actually uses the $(i+1)_{th}$ velocity for the i_{th} update. While the i_{th} velocity might just as easily have been used, it is neither better nor worse. Better is actually the average of the two.

G. C++ PROTOTYPE

The C++ version of the dynamic Aquarobot model is algorithmically identical to the LISP version. A complete source code listing with operating instructions may be found in Appendix D. As discussed in Chapter III, use of IRIS Performer structures required some modifications. For example, six-vectors were divided into pairs of three-vectors. The dynamic model is otherwise identical. A feature added to this version is the ability to pass spring and damper constants, drop height, and dynamic update time increment into main as command line arguments. After handling these optional arguments, main initializes Performer, instantiates and initializes dynamic and graphic Aquarobot objects, then cycles in an update-render loop.

The "graphic Aquarobot" object was not required in the LISP version as the "dynamic Aquarobot" slot values were directly accessed to render a stick figure. This version, however, draws thousands of filled polygons each cycle to render a single frame. IRIS Performer was used, as previously stated, primarily for its high performance in this task. The graphic Aquarobot is a hierarchical database containing the information required to draw Aquarobot. After each dynamic update, the body's position and orientation and the leg's joint angles are passed into "Dynamic Coordinate Systems" in the database prior to calling the draw routine.

H. SUMMARY

This chapter develops a simplified dynamic simulation model for Aquarobot using springs and dampers to provide the joint torques. Inverse kinematics, Jacobian matrices and their utility, and forces acting on Aquarobot are discussed. The model was implemented and tested using both LISP and C++. In the next chapter, simulation results of the spring and damper model, and also of the joint actuator model from Chapter IV, are presented.

VI. SIMULATION RESULTS

A. INTRODUCTION

This chapter presents the simulation results. The joint actuator simulation model was tested to verify the model and to experimentally determine suitable amplifier gains. As stated in the previous chapter, the spring and damper Aquarobot model was tested by dropping the robot from a low height and observing is dynamic behavior.

B. JOINT ACTUATOR SIMULATION

The motor class is designed such that constructor arguments are parameters listed on the motor specification sheet. At the time of the simulation, specification sheets for the motors used in Aquarobot were not available; therefore, another model was used. The model was tested by applying rated voltage to the motor and observing its acceleration and attained RPM. In Figure 6.1, both motor and output shaft, after 200:1 reduction gear, speeds and positions are shown. Motor scales are on the left, while output shaft scales are on the right. Qualitatively, the motor model is well behaved, and quantitatively, it closely matches the no-load speed parameter listed in the specifications.

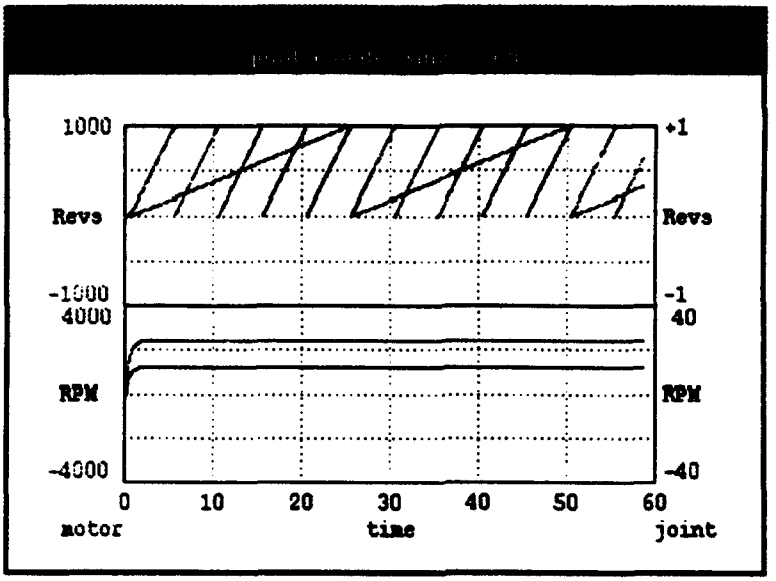


Figure 6.1
No Load Joint Actuator Response With +75VDC Applied to the Motor

To obtain fast joint actuator response to input orders, it is desirable to set the error signal, D/A converter output, gain to a relatively high value. A gain of 150 resulted in the optimum response. Values from 100 to 200 gave satisfactory results while higher values progressively reduced the effectiveness of the degenerative feedback. Figures 6.2 through 6.5 show joint actuator responses to the following sequence of shaft positioning orders, in revolutions, with various driver gain values: (+1/4, +1/2, -1, -1/2, +1, +3). Figure 6.2 illustrates the response with an error signal gain of 150 and with velocity and current feedback signals disabled.

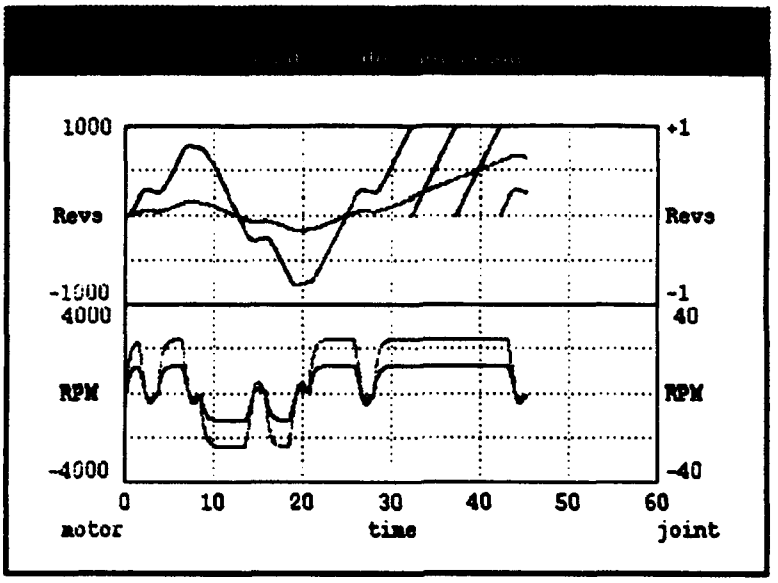


Figure 6.2
Joint Actuator Response with Error Signal Gain of 150.
Velocity and Current Feedback Disabled.

The optimum velocity feedback signal gain is a compromise between response time and stability. Too high a value results in slower response, while lower values allow increased overshoot. Values from three to seven were satisfactory. A gain of five proved optimum when combined with current feedback. Figure 6.3 displays the results of using various gain values for velocity feedback with current feedback still disabled.

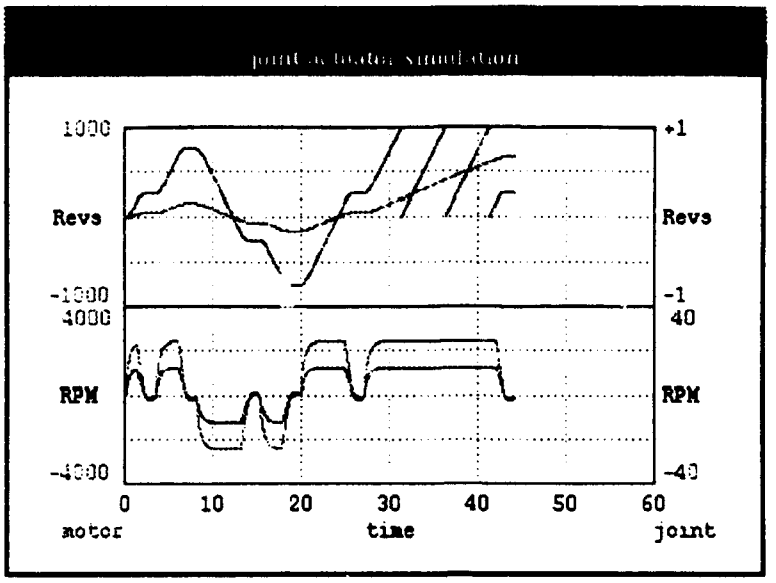


Figure 6.3a
Joint Actuator Response with Velocity Feedback Gain of 3 and Error Signal Gain of 150. Current Feedback Disabled.

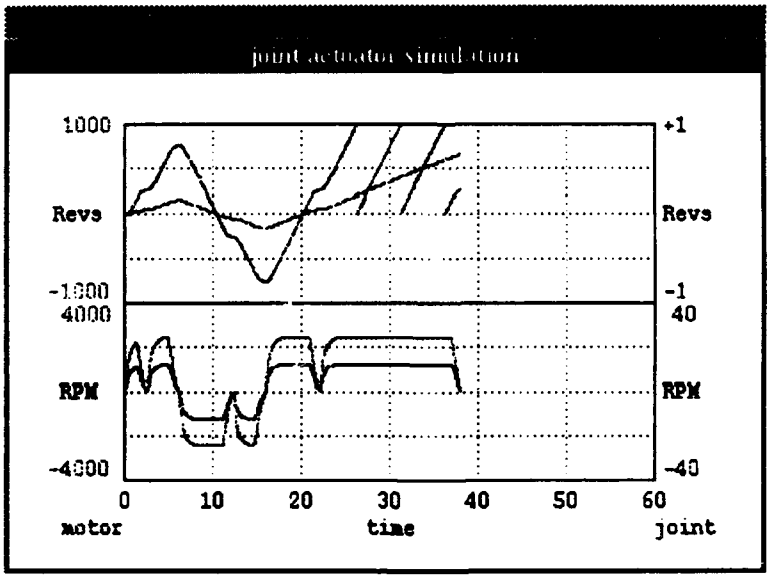


Figure 6.3b
Joint Actuator Response with Velocity Feedback Gain of 5 and Error Signal Gain of 150. Current Feedback Disabled.

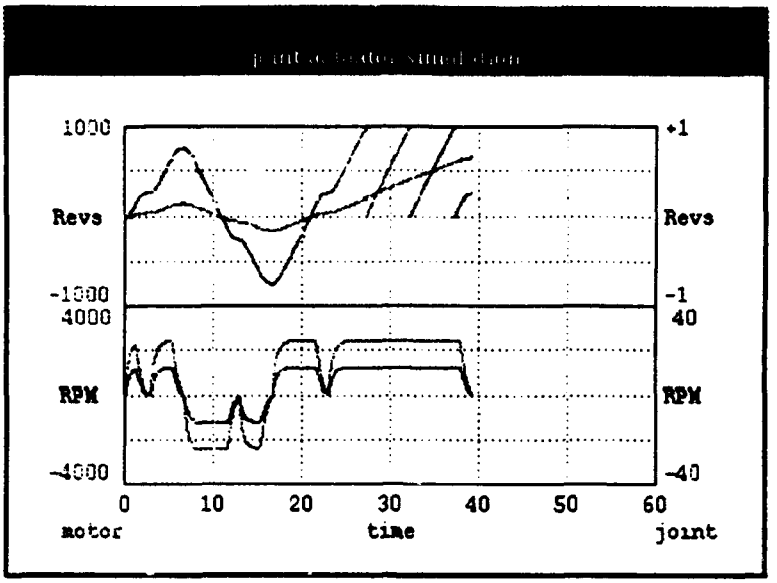


Figure 6.3c
Joint Actuator Response with Velocity Feedback Gain of 7 and Error Signal Gain of 150. Current Feedback Disabled.

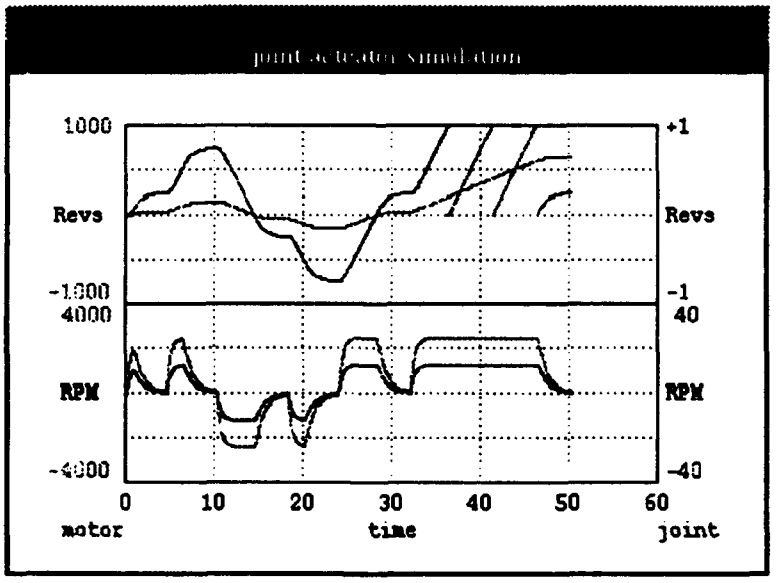


Figure 6.3d
Joint Actuator Response with Velocity Feedback Gain of 20 and Error Signal Gain of 150. Current Feedback Disabled.

The current feedback gain value turned out to be the most sensitive. A value of 0.3 produced negligible change, while 0.7 resulted in some oscillation. This regenerative feedback, intended to reduce response time, also aided in reducing the overshoot. As the error signal approached zero with speed on, back EMF caused a decelerating current which was aided by the regenerative current feedback. Figure 6.4 displays the results of using various gain values for current feedback with velocity feedback again disabled. In addition, in the presence of the degenerative velocity feedback, the error signal was overcome earlier by decelerating signals, bringing the current feedback contribution in even sooner. Figure 6.5 illustrates results with both velocity and current feedbacks in effect. The response of the joint actuator in this simulation highlights the effectiveness of the design: high error signal gain with velocity and current compensating feedback networks.

While these tests results are not for the specific motors used in Aquarobot, they provide a good set of gain parameters to use as a starting point in the experimental determination of the gains for those motors.

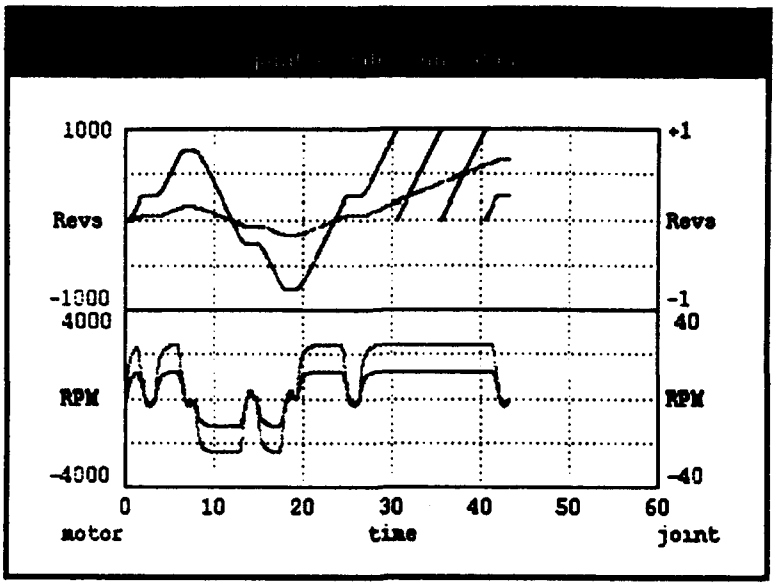


Figure 6.4a
Joint Actuator Response with Current Feedback Gain of 0.3 and Error Signal Gain of 150. Velocity Feedback Disabled.

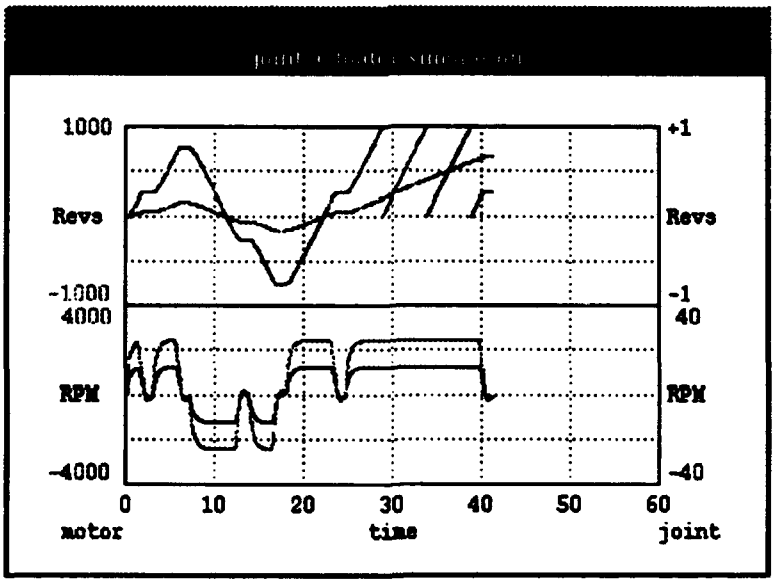


Figure 6.4b
Joint Actuator Response with Current Feedback Gain of 0.5 and Error Signal Gain of 150. Velocity Feedback Disabled.

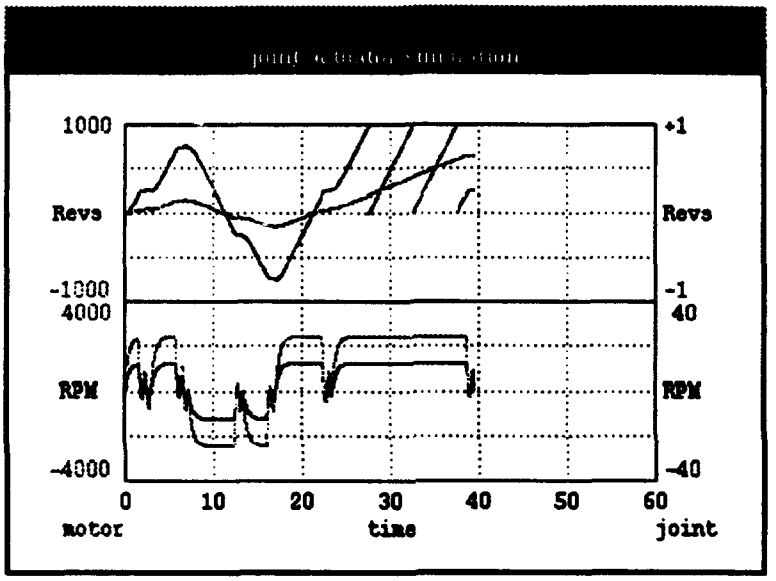


Figure 6.4c
Joint Actuator Response with Current Feedback Gain of 1.0 and Error Signal Gain of 150. Velocity Feedback Disabled.

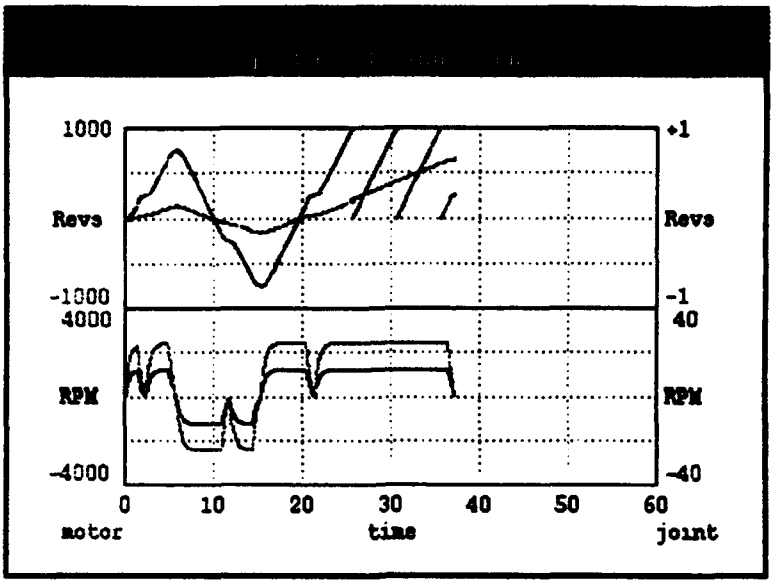


Figure 6.5a
Joint Actuator Response with Velocity Feedback Gain of 3, Error Signal Gain of 150, and Current Feedback Gain of 0.5.

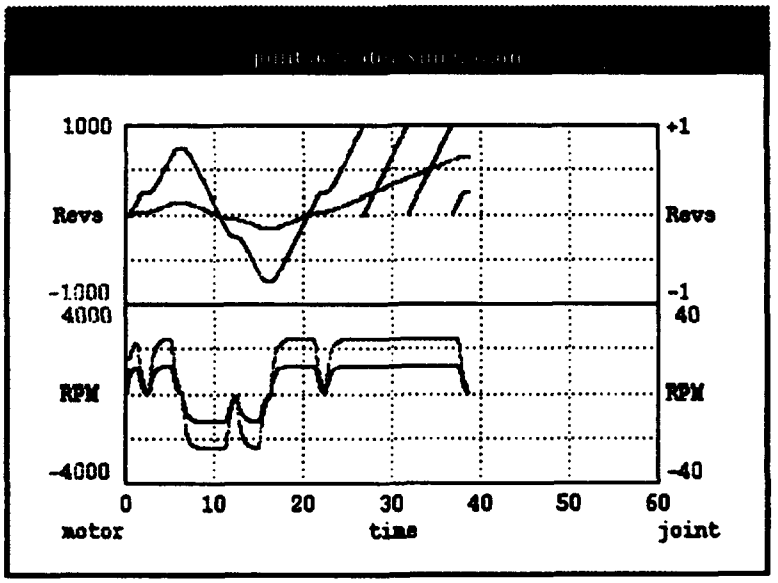


Figure 6.5b
Joint Actuator Response with Velocity Feedback Gain of 5, Error Signal Gain of 150, and Current Feedback Gain of 0.5.

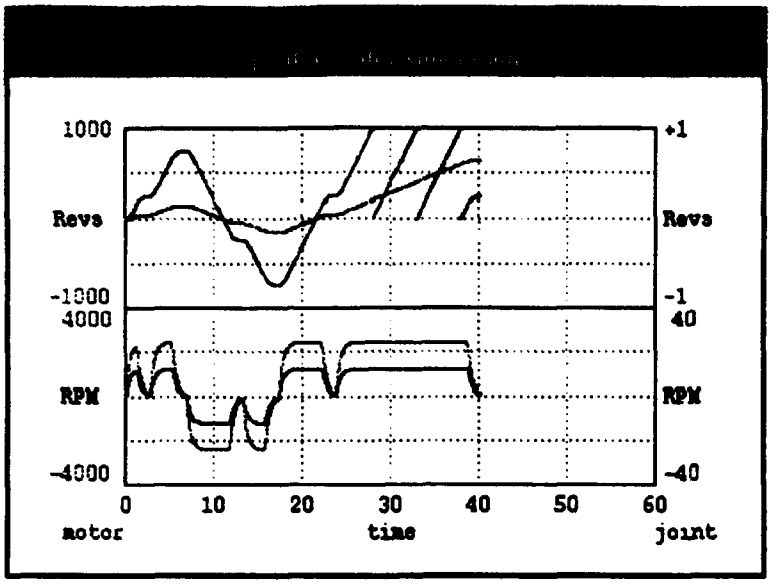


Figure 6.5c
Joint Actuator Response with Velocity Feedback Gain of 7, Error Signal Gain of 150, and Current Feedback Gain of 0.5.

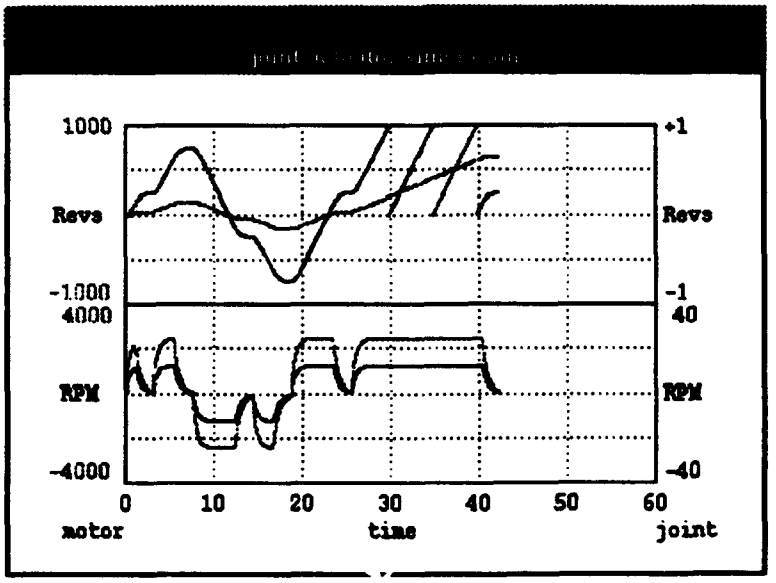


Figure 6.5d
Joint Actuator Response with Velocity Feedback Gain of 10, Error Signal Gain of 150, and Current Feedback Gain of 0.5.

C. AQUAROBOT SPRING AND DAMPER SIMULATION

LISP and C++ versions of the spring and damper Aquarobot model were tested by using a "droptest" in which the model is dropped from low height. It may be tilted, but not so much that it does not land on its feet. The LISP version served as the prototype, and after successful testing, the model was translated into C++.

The LISP simulation ran uncompiled on a Sun Sparc-10 at six to eight frames per minute and was too slow for comprehension of motion detail. It did, however, allow experimental determination of spring and damper constants sufficient to support the model when dropped. To increase the simulation speed, two dynamic update cycles were run between each display, and the source code was then compiled. After compilation, the simulation ran at near 30 frames per minute, with 60 dynamic updates of 50ms each, to achieve a simulation with approximately a 10:1 time dilation. This simulation was fast enough for an observer to assimilate the dynamic behavior of the model which was qualitatively satisfactory and considered successful. Some additional fine tuning of experimentally determined parameters was done prior to translation to C++.

After translation to C++/Performer, the model was again tested, and a real-time simulation was achieved on a four processor IRIS 440/RE workstation. Only three processors were actually utilized, one assigned to each of the following tasks: application, database pre-draw cull, and database rendering traversal. The application processor utilization was approximately fifteen percent which indicates that the increased complexity of adding the joint actuators will not present any difficulty. The cull processor utilization

was approximately twenty percent while that of the rendering processor veried from fifty to seventy percent. (Note: the source code was not compiled and/or linked with optimizations on.) Typical images obtained can be found in [Goetz, 1994].

Running in the three processor configuration described above, Performer synchronized the framerate to the fixed 50ms dynamic updates by limiting the framerate to 20Hz. On a single processor IRIS R-4000, where the application, database cull, and database rendering were forced to run sequentially, the highest framerate achieved was 10Hz. This resulted in a 2:1 time dilation (slow down) simulation.

D. SUMMARY

Testing of both simulation models was considered successful. While it is unfortunate that there was insufficient time to incorporate the joint actuator model into the Aquarobot model, replacing the springs and dampers, the topic was given some time and effort. This next step is among the topics addressed in the final chapter: Summary and Conclusions.

VII. SUMMARY AND CONCLUSIONS

A. UTILITY OF LISP FOR EXPERIMENTAL PROGRAMMING

The primary benefit of using LISP as a prototyping language in this thesis was the immediate testing capability it provided. No test routines were required. Each function was easily tested by direct calls as it was developed. While compilation capability allowed a faster simulation, repeated compilations were not required as the routines could called from the interpreter's command line. Finally, nesting functions allowed larger and larger integrated blocks of source code to be tested.

One of the benefits of using LISP during prototyping was realized when an apparent limit cycle appeared if Aquarobot was tilted when dropped. The problem was actually a lack of rotational damping due to the absence of the omega cross r term in the foot velocity calculation. Without it, damping ceased shortly after landing because of "zero translational foot velocity." While the author took a considerable period of time to find the cause of the problem, with a simple modification to the LISP source code, the correction was quickly and easily verified.

B. INCORPORATING THE JOINT ACTUATOR MODEL

The next step required for the Aquarobot dynamic simulation model is to replace the springs and dampers with joint actuators. This was initially assumed to be a simpler task

than it turned out to be. The difficulty arises from a conflict over control of joint state variables. In the spring and damper model, the joints only supply state dependent torques which are then used to dynamically update the robot's body. After the body is updated, the new states of the joints depend only on the new body position and not on the joint torques. In the joint actuator model, the joint state depends directly upon the motor torque. Two possible solutions are proposed to eliminate the conflict.

One possible solution is to extend the massless leg simplification to the motor and gear-train, making them inertialess. While this seems easiest and will probably have as little impact as the massless legs, the overall effect may be greater than anticipated due to the large reduction ratios involved. Recall that motor inertia reflected outside the reduction gear is multiplied by the square of the reduction ratio. A C++ version of the joint actuator model is included in Appendix B. This model is a variation of the original LISP version and provides state dependent torques as output rather than the state itself. It could be used to implement inertialess joint actuators, and the changes required in the Aquarobot model would be minor.

Another possible solution is to use the concept of "added mass," an apparent increase in mass (affecting acceleration) due to the internal inertia of the drive motors. Assuming a unit acceleration for one of the body's six degrees of freedom, it is possible to calculate the resulting joint accelerations. If a joint acceleration is known, inertial torque in the joint can be calculated. The net torque for the joint then is armature torque (determined using motor applied voltage and speed) minus the inertial torque. Doing this for all six degrees

of freedom gives a matrix of added mass which can be inverted to get acceleration for any given vector of joint motor applied voltages. An "equilibrium torque," torque vector which results in a zero joint acceleration vector, is also needed and must be calculated [Koozekanani, 1983].

C. SUMMARY

IRIS Performer has proven its utility as a graphics rendering tool in a real time simulation. It also provided easy synchronization for fixed duration integration intervals.

A real-time dynamic simulation of Aquarobot was accomplished and is eventually expected to provide a valuable tool for Aquarobot control software developers. While an integrated model, with the joint actuators in place, is not yet completed, we are a step closer, and the task has certainly proven to be feasible. Once the joint actuators are installed into the Aquarobot model, simple walking simulations, on smooth, flat terrain may be achieved. Concurrent work on collision detection for uneven terrain will also further improve the simulation model when incorporated [Goetz, 1994].

Further work could improve on the Performer Aquarobot rendering database to decrease the rendering time. This is the area where there is the most room for improvement in cycle time. Finally, other than using the "faster" Performer routines, no attempt has been made to optimize the source code which was written with ease of modification in mind. Utilization of compile and link optimizations, and eventually some

source code tuning, may also contribute toward achievement of a real-time Aquarobot simulation on a single processor.

APPENDIX A

LOADING AND OPERATING INSTRUCTIONS

To run demo, start LISP Interpreter and call:
    (load "joint-loader")

SOURCE CODE FILES

; "joint-loader"

```
; load files
(load "math.routines.cl")
(load "time.routines.cl")
(load "diff-counter.class.cl")
(load "amplifier-clipper.class.cl")
(load "shaft.class.cl")
(load "motor.class.cl")
(load "reduction-gear.class.cl")
(load "joint.class.cl")
(load "joint.instance.cl")
(load "window.instance.cl")

; execute tests
(move-joint-mult -.25 4)
(move-joint-mult .25 4)
(move-joint-mult .05 4)
(move-joint-list '(.25 .5 -1 -.5 1 3))
(run-joint 15)
```

; "diff-counter.class.cl"

```
(defclass diff-counter ()
  ((current-count
     :accessor current-count
     :initform 0
     :type     float )))

(defmethod diff-signal ((dc diff-counter) plus-input minus-input)
  (setf (current-count dc)
      (+ (current-count dc) plus-input (- minus-input))))
```

; "amplifier-clipper.class.cl"

```
(defclass amplifier-clipper ()
  ((amplification-factor
     :initarg :amplification-factor
     :accessor amplification-factor
     :initform 1
     :type     float)
   (max-value
     :initarg :max-value
     :accessor max-value
     :initform 1
     :type     float)
   (min-value
     :initarg :min-value
     :accessor min-value
     :initform -1
     :type     float)))

(defmethod amplify ((amp amplifier-clipper) input-value)
  (max (min (* (amplification-factor amp) input-value)
         (max-value amp))
     (min-value amp)))
```

```
(defclass motor-driver (amplifier-clipper)
  ((displacement-gain
    :initarg :displacement-gain
    :accessor displacement-gain
    :initform 1
    :type    float)
   (velocity-fb-gain
    :initarg :velocity-fb-gain
    :accessor velocity-fb-gain
    :initform 0
    :type    float)
   (current-fb-gain
    :initarg :current-fb-gain
    :accessor current-fb-gain
    :initform 0
    :type    float)))

(defmethod drive((driver motor-driver) displacement-input
                                       velocity-input
                                       current-input)
  (amplify driver (+ (* displacement-input (displacement-gain driver))
                     (* (- velocity-input) (velocity-fb-gain  driver))
                     (* current-input     (current-fb-gain   driver)))))
```

```
; "shaft.class.cl"


(defclass shaft ()
  ((angular-position        ; radians
    :accessor theta
    :initform 0
    :type     float)
   (angular-velocity        ; rad/sec
    :accessor omega
    :initform 0
    :type     float)
   (inertia                 ; Kg-(meters-square)
    :initarg :I
    :accessor I
    :initform 0
    :type     float)
   (coulomb-friction-constant ; Newton-meters (Fc >= 0)
    :initarg :Fc
    :accessor Fc
    :initform 0
    :type     float)
   (viscous-friction-constant ; Newton-meters/(rad/sec) (Fv >= 0)
    :initarg :Fv
    :accessor Fv
    :initform 0
    :type     float)
   (time-stamp              ; seconds
    :accessor time-stamp
    :initform (system-time))))

(defmethod set-shaft ((s shaft) theta omega)
  (setf (omega s) omega)
  (setf (theta s) theta))

(defmethod reset-shaft ((s shaft))
  (set-shaft s 0 0)
  (setf (time-stamp s) (system-time)))

(defmethod connect ((source shaft) (load shaft))
  (setf (time-stamp load) (time-stamp source))
  (set-shaft load (theta source) (omega source)))
```

; "motor.class.cl"

```
(defclass motor (shaft)
  ((torque-constant    ; Newton-meters/ampere
    :initarg :Kt
    :accessor Kt
    :type    float)
   (back-emf-constant  ; Volts/(rad/sec)
    :initarg :Kb
    :accessor Kb
    :type    float)
   (armature-resistance ; ohms (must be > 0)
    :initarg :R
    :accessor R
    :initform 1
    :type    float)
   (max-brush-drop     ; volts
    :initarg :max-brush-drop
    :accessor max-brush-drop
    :initform 2.0
    :type    float)
   (half-brush-drop-source-value ; volts
    :initarg :half-BD-value
    :accessor half-BD-value
    :initform 3.0
    :type    float)
   (armature-current   ; amperes (saved for feedback purposes)
    :accessor armature-current
    :initform 0
    :type    float)))

(defmethod applied-voltage ((m motor) source-voltage)
  (if (zerop source-voltage) 0
    (* source-voltage
      (- 1
        (* (/ (max-brush-drop m) (abs source-voltage))
          (- 1
            (exp (* (log 0.5)
                  (/ (abs source-voltage)
                    (half-BD-value m)))))))))))

(defmethod developed-torque ((m motor) source-voltage)
  (setf (armature-current m)
      (/ (- (applied-voltage m source-voltage) (* (Kb m) (omega m))) (R m)))
  (* (Kt m) (armature-current m)))
```

```
(defmethod omega-dot ((m motor) source-voltage load-inertia load-torque
                      load-coulomb-friction-constant
                      load-viscous-friction-constant)
  (let* ((torque (+ (developed-torque m source-voltage) load-torque))
         (Fc-total (+ (Fc m) load-coulomb-friction-constant))
         (friction-loss
           (if (zerop (omega m))
               (if (zerop torque)
                   0
                   (if (> Fc-total (abs torque))
                       torque
                       (* Fc-total (sgn torque))))
               (+ (* (+ (Fv m) load-viscous-friction-constant) (omega m))
                  (* Fc-total (sgn (omega m))))))))
    (/ (- torque friction-loss) (+ (I m) load-inertia))))

(defmethod run-motor ((m motor) source-voltage load-inertia load-torque
                      load-coulomb-friction-constant
                      load-viscous-friction-constant)
  (let ((dt (delta-time (time-stamp m)))
        (omega-dot (omega-dot m source-voltage load-inertia load-torque
                      load-coulomb-friction-constant
                      load-viscous-friction-constant)))
    (setf (theta m) (+ (theta m) (* (omega m) dt)))
    (setf (omega m) (+ (omega m) (* omega-dot dt)))
    (setf (time-stamp m) (+ (time-stamp m) dt))))
```

```
; "reduction-gear.class.cl"


(defclass reduction-gear ()
  ((gear-ratio
    :initarg  :gear-ratio
    :accessor  gear-ratio
    :initform  1
    :type      float)
   (in-shaft
    :initarg  :in-shaft
    :accessor  in-shaft
    :initform  (make-instance 'shaft))
   (out-shaft
    :initarg  :out-shaft
    :accessor  out-shaft
    :initform  (make-instance 'shaft))))

(defmethod rg-reduce ((rg reduction-gear) value)
  (/ value (gear-ratio rg)))

(defmethod rg-reflect ((rg reduction-gear) value)
  (* value (gear-ratio rg)))

(defmethod rg-inertia-forward ((rg reduction-gear) inertia-value)
  (* inertia-value (sqr (gear-ratio rg))))

(defmethod rg-inertia-backward ((rg reduction-gear) inertia-value)
  (/ inertia-value (sqr (gear-ratio rg))))

(defmethod rg-connect ((rg reduction-gear))
  (set-shaft (out-shaft rg)
        (rg-reduce rg (theta  (in-shaft rg)))
        (rg-reduce rg (omega  (in-shaft rg)))))
```

```
; "joint.class.cl"


(defclass joint ()
  ((pulse-counter
    :initarg  :pc
    :accessor pc
    :initform (make-instance 'diff-counter))
   (driver
    :initarg  :driver
    :accessor driver
    :initform (make-instance 'motor-driver))
   (prime-mover
    :initarg  :prime-mover
    :accessor prime-mover
    :initform (make-instance 'motor))
   (red-gear
    :initarg  :red-gear
    :accessor red-gear
    :initform (make-instance 'reduction-gear))
   (load-shaft
    :initarg  :load-shaft
    :accessor load-shaft
    :initform (make-instance 'shaft))))

(defmethod motor-load-inertia ((j joint))
  (+ (I (in-shaft (red-gear j)))
    (rg-inertia-backward (red-gear j) (+ (I (out-shaft (red-gear j)))
                                         (I (load-shaft j))))))

(defmethod motor-load-coulomb-friction-constant ((j joint))
  (+ (Fc (in-shaft (red-gear j)))
    (rg-reduce (red-gear j) (+ (Fc (out-shaft (red-gear j)))
                               (Fc (load-shaft j))))))

(defmethod motor-load-viscous-friction-constant ((j joint))
  (+ (Fv (in-shaft (red-gear j)))
    (rg-reduce (red-gear j) (+ (Fv (out-shaft (red-gear j)))
                               (Fv (load-shaft j))))))

(defmethod feedback ((j joint))
  (- (theta (out-shaft (red-gear j))) (theta (load-shaft j))))
```

```
(defmethod increment-joint ((j joint) order)
  (run-motor (prime-mover j)
        (drive (driver j) (diff-signal (pc j) order (feedback j))
                  (* .003 (RAD/SECtoRPM (omega (prime-mover j))))
                  (armature-current (prime-mover j)))
        (motor-load-inertia j)
        0 ; load not producing any torque
        (motor-load-coulomb-friction-constant j)
        (motor-load-viscous-friction-constant j))
  (connect (out-shaft (red-gear j)) (load-shaft j))
  (connect (prime-mover j) (in-shaft (red-gear j)))
  (rg-connect (red-gear j)))

(defmethod step-input-to-joint ((j joint) step)
  (diff-signal (pc j) (REVtoRAD step) 0))

(defmethod reset-joint ((j joint))
  (setf (current-count (pc j)) 0)
  (reset-shaft (load-shaft j))
  (reset-shaft (out-shaft (red-gear j)))
  (reset-shaft (in-shaft (red-gear j)))
  (reset-shaft (prime-mover j)))
```

```
; "joint.instance.cl"


(setf joint1 (make-instance 'joint
        :driver     (make-instance 'motor-driver
                    :amplification-factor  1
                    :max-value          75   ; volts
                    :min-value         -75   ; volts
                    :displacement-gain   150
                    :velocity-fb-gain      5
                    :current-fb-gain      .5 )
        :prime-mover (make-instance 'motor
                    :I          .0005  ; Kg-m*m
                    :Fc          .0075  ; N-m
                    :Fv          .00004 ; N-m/(rad/sec)
                    :Kt          .005   ; N-m/ampere
                    :Kb          .255   ; Volts/(rad/sec)
                    :R           1   ) ; ohms
        :red-gear    (make-instance 'reduction-gear
                    :gear-ratio 200)
        :load-shaft  (make-instance 'shaft
                    :I        5      ; Kg-m*m
                    :Fc       .1     ; N-m
                    :Fv       .02  ))) ; N-m/(rad/sec)

(defun move-joint (delta-theta)
  (setf (time-stamp (prime-mover joint1)) (system-time))
  ;input delta-theta a little at a time
  (if (< delta-theta 0)
     ;negative delta-theta (use -0.015 steps)
     (do* ((input-index 0 (+ input-index 0.015)))
        ((< (+ delta-theta input-index) 0.015)
         (step-input-to-joint joint1 (+ delta-theta input-index)))
        (step-input-to-joint joint1 -0.015)
        (increment-joint joint1 0)
        (display-state *display* joint1 (time-stamp (prime-mover joint1))))
     ;positive delta-theta (use 0.015 steps)
     (do* ((input-index 0 (+ input-index 0.015)))
        ((< (- delta-theta input-index) 0.015)
         (step-input-to-joint joint1 (- delta-theta input-index)))
        (step-input-to-joint joint1 0.015)
        (increment-joint joint1 0)
        (display-state *display* joint1 (time-stamp (prime-mover joint1)))))
  ;delta-theta entry into PC is complete
  ;cycle until ordered position is reached
  (do* ((index1))
     ((and (< (abs (current-count (pc joint1))) 0.05)
          (< (abs (omega (prime-mover joint1))) 10)) (pprint 'stop))
     (increment-joint joint1 0)
     (displa; -state *display* joint1 (time-stamp (prime-mover joint1)))))
```

```
(defun move-joint-mult (delta-theta mult)
  (clear-and-reset)
  (dotimes (i mult) (move-joint delta-theta)))

(defun run-joint (speed)
  (clear-and-reset)
  (do* ((dtime (delta-time (time-stamp (prime-mover joint1)))
               (delta-time (time-stamp (prime-mover joint1)))))
       ((< (abs (- (RPMtoRAD/SEC speed) (omega (load-shaft joint1))))
           0.1) (pprint 'stop))
       (increment-joint joint1 (* dtime (RPMtoRAD/SEC speed)))
       (display-state *display* joint1 (time-stamp (prime-mover joint1)))))

(defun move-joint-list (delta-theta-list)
  (clear-and-reset)
  (if (equalp nil delta-theta-list) nil
      (move-joint-list-2 delta-theta-list)))

(defun move-joint-list-2 (delta-theta-list)
  (move-joint (car delta-theta-list))
  (if (equalp nil (cdr delta-theta-list)) nil
      (move-joint-list-2 (cdr delta-theta-list))))

(defun clear-and-reset ()
  (update-minutes *display* 0)
  (reset-system-time)
  (reset-joint joint1))
```

; "window.instance.cl"

```
; dimensions for x-y coord system (window size auto adjusts)
(setf *x-origin*   50)
(setf *x-length*  500)
(setf *x-tics*      6)
(setf *y-origin*   50)
(setf *y-length*  340)
(setf *y-tics*      8)

(setf *max-speed*  4000) ; (max rpm's of motor scale)
(setf *max-revs*   1000) ; (max rev's of motor scale)

(require :xcw)
(use-package :cw)
(cw:initialize-common-windows)
```

```
(defmethod draw-grid ((window window-stream))
  (draw-line-xy window *x-origin*                    ;top border
                (+ *y-origin* *y-length*)
                (+ *x-origin* *x-length*)
                (+ *y-origin* *y-length*))
  (draw-line-xy window *x-origin*                    ;middle x axis
                (+ *y-origin* (/ *y-length* 2))
                (+ *x-origin* *x-length*)
                (+ *y-origin* (/ *y-length* 2)))
  (draw-line-xy window *x-origin*                    :bottom border
                *y-origin*
                (+ *x-origin* *x-length*)
                *y-origin*)
  (draw-line-xy window *x-origin*                    :left border
                *y-origin*
                *x-origin*
                (+ *y-origin* *y-length*))
  (draw-line-xy window (+ *x-origin* *x-length*)     ;right border
                *y-origin*
                (+ *x-origin* *x-length*)
                (+ *y-origin* *y-length*))
  (dotimes (i *x-tics*)                              ; mark x axis
    (draw-line-xy window (+ *x-origin*
                            (/ (* i *x-length*) *x-tics*))
                          *y-origin*
                          (+ *x-origin*
                            (/ (* i *x-length*) *x-tics*))
                          (+ *y-origin* *y-length*)
                          :dashing '(1 3)))
  (dotimes (i *y-tics*)                              ; mark y axis
    (draw-line-xy window *x-origin*
                          (+ *y-origin*
                            (/ (* i *y-length*) *y-tics*))
                          (+ *x-origin* *x-length*)
                          (+ *y-origin*
                            (/ (* i *y-length*) *y-tics*))
                          :dashing '(1 3)))
  (label-graph window))
```

```
(defmethod label-graph ((w window-stream))
  ; theta labels
  (setf (window-stream-y-position w) (+ *y-origin* *y-length* -4))
  (setf (window-stream-x-position w) (+ *x-origin* -35))
  (setf (window-stream-foreground-color w) red)
  (format w "~s" *max-revs*)
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 5))
  (setf (window-stream-foreground-color w) blue)
  (format w "+1")
  (setf (window-stream-y-position w) (+ *y-origin* (* *y-length* 0.75) -3))
  (setf (window-stream-x-position w) (+ *x-origin* -40))
  (setf (window-stream-foreground-color w) black)
  (format w "Revs")
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 5))
  (format w "Revs")
  (setf (window-stream-y-position w) (+ *y-origin* (* *y-length* 0.5) 3))
  (setf (window-stream-x-position w) (+ *x-origin* -43))
  (setf (window-stream-foreground-color w) red)
  (format w "~s" (- *max-revs*))
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 5))
  (setf (window-stream-foreground-color w) blue)
  (format w "-1")
  ; omega labels
  (setf (window-stream-y-position w) (+ *y-origin* (* *y-length* 0.5) -10))
  (setf (window-stream-x-position w) (+ *x-origin* -35))
  (setf (window-stream-foreground-color w) red)
  (format w "~s" *max-speed*)
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 10))
  (setf (window-stream-foreground-color w) blue)
  (format w "~s" (/ *max-speed* 100))
  (setf (window-stream-y-position w) (+ *y-origin* (* *y-length* 0.25) -3))
  (setf (window-stream-x-position w) (+ *x-origin* -36))
  (setf (window-stream-foreground-color w) black)
  (format w "RPM")
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 5))
  (format w "RPM")
  (setf (window-stream-y-position w) (+ *y-origin* 3))
  (setf (window-stream-x-position w) (+ *x-origin* -43))
  (setf (window-stream-foreground-color w) red)
  (format w "~s" (- *max-speed*))
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* 5))
  (setf (window-stream-foreground-color w) blue)
  (format w "~s" (- (/ *max-speed* 100)))
  ; time labels
  (setf (window-stream-foreground-color w) black)
  (setf (window-stream-y-position w) (+ *y-origin* -15))
  (setf (window-stream-x-position w) (+ *x-origin* -3))
  (format w "0")
  (setf (window-stream-x-position w) (+ *x-origin* (/ *x-length* 6) -7))
  (format w "10")
  (setf (window-stream-x-position w) (+ *x-origin* (/ *x-length* 3) -7))
```

```
  (format w "20")
  (setf (window-stream-x-position w) (+ *x-origin* (/ *x-length* 2) -7))
  (format w "30")
  (setf (window-stream-x-position w) (+ *x-origin* (* *x-length* (/ 2 3)) -7))
  (format w "40")
  (setf (window-stream-x-position w) (+ *x-origin* (* *x-length* (/ 5 6)) -7))
  (format w "50")
  (setf (window-stream-x-position w) (+ *x-origin* *x-length* -7))
  (format w "60")
  (setf (window-stream-y-position w) (+ *y-origin* -30))
  (setf (window-stream-x-position w) (+ *x-origin* -35))
  (setf (window-stream-foreground-color w) red)
  (format w "motor")

  (setf (window-stream-x-position w) (+ *x-origin* *x-length*))
  (setf (window-stream-foreground-color w) blue)
  (format w "joint")
  (setf (window-stream-foreground-color w) black)
  (setf (window-stream-x-position w) (+ *x-origin* (* *x-length* .5) -13))
  (format w "time"))

(defmethod draw-motor-position ((window window-stream) revolutions)
  (draw-point-xy window
          *x-time-value*
          (+ *y-origin* (* 0.75 *y-length*)
            (* 0.25 *y-length*
              (cond ((zerop revolutions) 0)
                  ((< revolutions 0)
                    (/ (- (mod revolutions *max-revs*)
                          *max-revs*)
                       *max-revs*))
                  (t (/ (mod revolutions *max-revs*)
                        *max-revs*)))))
          :color red))

(defmethod draw-load-position ((window window-stream) revolutions)
  (draw-point-xy window
          *x-time-value*
          (+ *y-origin* (* 0.75 *y-length*)
            (* 0.25 *y-length*
              (cond ((zerop revolutions) 0)
                  ((< revolutions 0)
                    (- (mod revolutions 1.0) 1.0))
                  (t (mod revolutions 1.0)))))
          :color blue))
```

```
(defmethod draw-motor-speed ((window window-stream) speed)
  (draw-point-xy window
                 *x-time-value*
                 (+ *y-origin* (* 0.25 *y-length*)
                   (* 0.25 *y-length* (/ speed *max-speed*)))
                 :color red))

(defmethod draw-load-speed ((window window-stream) speed)
  (draw-point-xy window
                 *x-time-value*
                 (+ *y-origin* (* 0.25 *y-length*)
                   (* 0.25 *y-length* 100 (/ speed *max-speed*)))
                 :color blue))

(defmethod update-minutes ((window window-stream) minutes)
  (clear window)
  (draw-grid window)
  (setf *minutes* minutes))

(defmethod set-x-coord ((window window-stream) seconds)
  (if (> (truncate (/ seconds 60)) *minutes*)
     (update-minutes window (truncate (/ seconds 60))))
  (setf *x-time-value* (round (+ *x-origin*
                                 (* (/ (mod seconds 60) 60)
                                   *x-length*)))))

(defmethod display-state ((window window-stream) (j joint) current-time)
  (set-x-coord window current-time)
  (draw-motor-speed window (RAD/SECtoRPM (omega (prime-mover j))))
  (draw-load-speed window (RAD/SECtoRPM (omega (load-shaft j))))
  (draw-motor-position window (RADtoREV (theta (prime-mover j))))
  (draw-load-position window (RADtoREV (theta (load-shaft j)))))

(setf *display*
  (make-window-stream
    :left 1
    :bottom 1
    :width (+ *x-length* (* 2 *x-origin*))
    :height (+ *y-length* (* 2 *y-origin*))
    :background-color white
    :foreground-color black
    :inner-region-left
    :inner-region-bottom
    :inner-region-width
    :inner-region-height
    :title "joint actuator simulation"
    :activate-p t))

(setf *minutes* 0)
(draw-grid *display*)
```

; "time.routines.cl"

```
(defun reset-system-time ()
  (setf *RefTime* (get-internal-real-time)))

(defun system-time ()
  (/ (- (get-internal-real-time) *RefTime*) 1000.0))

(defun delta-time (time)
  (- (system-time) time))

(reset-system-time)
```

; "math.routines.cl"

```
(defun sqr (x) (* x x))

(defun sgn (x)
  (if (< x 0) -1 1))

(defun RPMtoRAD/SEC (rpm)
  (* rpm (/ pi 30))) ; * 2pi/60

(defun RAD/SECtoRPM (rad/sec)
  (* rad/sec (/ 30 pi)))

(defun REVtoRAD (rev)
  (* 2 pi rev))

(defun RADtoREV (rad)
  (/ rad (* 2 pi)))
```

APPENDIX B

SOURCE CODE FILES (untested)

// file "motor.h"

```
#ifndef __MOTOR_H
#define __MOTOR_H

#include <math.h>
#include <stdio.h>

class motor {   // inertia-less motor class
private:
  float Fc;     // Coulomb friction constant
  float Fv;     // Viscous friction constant
  float Kt;     // Torque constant
  float Kb;     // Back EMF constant
  float Ra;     // Armature resistance
  float BDm;    // Rated brush brop value
  float BDc;    // Ln(1/2)/BDh
            // BDh = Voltage applied such that brush drop = 1/2 BDm
  // subtracts brush drop from source voltage
  float AppliedVoltage(float);

public:
  float Ia;     // Armature current (available for current feedback)
  motor() {}
  // called after constructor for initialization
  void init_motor(float TorqueConstant,          // N*m/Ampere
                  float BackEMFConstant,         // Volts/RPM
                  float NoLoadCurrent,           // Amperes
                  float NoLoadSpeed      = 1000.0, // RPM
                  float StartingCurrent  =    0.0, // Amperes
                  float ArmatureResistance =  1.0, // Ohms
                  float RatedBrushDrop   =    2.0, // Volts
                  float HalfBrushDrop    =    3.0);// Volts
  float DevelopedTorque(float, float);
};
```

```
// provide initialization for specific motor types in Aquarobot
void makeRA20(motor& m);
void makeRH25(motor& m);

#endif
/* EOF */
```

// file "motor.c"

```
#include "motor.h"

void motor::init_motor(float TorqueConstant,
                       float BackEMFConstant,
                       float NoLoadCurrent,
                       float NoLoadSpeed,
                       float StartingCurrent,
                       float ArmatureResistance,
                       float RatedBrushDrop,
                       float HalfBrushDrop)
{
  Ia = 0.0;
  Fc =  TorqueConstant * StartingCurrent;
  Fv = (TorqueConstant * NoLoadCurrent - Fc) / NoLoadSpeed;

  Kt = TorqueConstant;
  Kb = BackEMFConstant;
  Ra = ArmatureResistance;

  BDm = RatedBrushDrop;
  if (BDm < 0.0) {
    printf("error: rated brush drop must be >= 0 Volts...\n");
    printf("default brush drop value (0 Volts) used...\n");
    BDm = 0.0;
    BDc = 1.0;
  }
  else if (BDm == 0.0) {
    BDc = 1.0;
  }
  else if (HalfBrushDrop < BDm/2) {
    printf("error: half brush drop must be >= 1/2 rated brush drop...\n");
    printf("defaulted to 1/2 RatedBrushDrop...\n");
    BDc = log(0.5) * 2.0 / BDm;
  }
  else {
    BDc = log(0.5) / HalfBrushDrop;
  }
}
```

```
float motor::DevelopedTorque(float SourceVoltage, float omega)
{
 float Torque;
 float FrictionLoss;

 // determine armature current and save
 Ia = (AppliedVoltage(SourceVoltage) - (Kb * omega)) / Ra;

 // determine motor torque
 Torque = Kt * Ia;

 // calculate loss
 // loss opposes omega (viscous and coulomb components)
 if     (omega > 0.0) FrictionLoss = omega * Fv + Fc;
 else if (omega < 0.0) FrictionLoss = omega * Fv - Fc;
 // if (omega == 0) : loss opposes Torque (no viscous component)
 else if (Torque > Fc)  FrictionLoss =   Fc;
 else if (Torque < -Fc) FrictionLoss = - Fc;
 // if ((omega == 0) && (|Torque| < Fc)) : Torque insufficient to overcome Fc
 else FrictionLoss = MotorTorque;

 return (Torque - FrictionLoss);
}
```

```
/* Private Function */

float motor::AppliedVoltage(float Vs)
{
  // return Vs(1 - (BDm/|Vs|)*(1 - exp(ln(1/2)*|Vs|/BDh)))
  if (Vs == 0.0) {
    return 0.0;
  }
  else if (Vs < 0.0) { // negative Vs
    return (Vs + (BDm * (1 - exp(-BDc*Vs))));
  }
  else {              // positive Vs
    return (Vs - (BDm * (1 - exp( BDc*Vs))));
  }
}

/* specific Aquarobot motor type initializations */

// Aquarobot motor parameters for init_motor
// read from spec sheet provided; includes harmonic gear
#define RA20_PARAMETERS 32.0, 3.4, 0.78, 25.0, 0.32, 3.2, 2.0, 3.0
#define RH25_PARAMETERS 33.0, 3.5, 0.89, 25.0, 0.48, 1.1, 2.0, 3.0

void makeRA∠0(motor& m)
{
  m.init_motor(RA20_PARAMETERS);
}

void makeRH25(motor& m)
{
  m.init_motor(RH25_PARAMETERS);
}

/* EOF */
```

```
// file "amplifier_clipper.h"

#ifndef __AMP_CLIP_H__
#define __AMP_CLIP_H__

class amplifier_clipper {

private:
  float gain;
  float max_value;
  float min_value;

public:
  amplifier_clipper() {}
  // call after constructor for initialization
  void init_amplifier_clipper(float g, float max_v, float min_v);

  float amplify(float input_value);
};

class aqua_driver : private amplifier_clipper {
private:
  float theta_gain;
  float omega_gain;
  float current_gain;

public:
  aqua_driver() {}
  // call after constructor for initialization
  init_aqua_driver(float displacement_gain,
                   float velocity_gain,
                   float current_feedback_gain);
  float drive(float theta_error, float omega, float current);
};

// provide initialization for specific joint on aqualeg
void makeJ1driver(aqua_driver& d);
void makeJ2driver(aqua_driver& d);
void makeJ3driver(aqua_driver& d);

#endif
```

```
// file "amplifier_clipper.c"

#include "amplifier_clipper.h"

void
init_amplifier_clipper(float g, float max_v, float min_v)
{
 gain      = g;
 max_value = max_v;
 min_value = min_v;
}

float
amplifier_clipper::amplify(float input_value)
{
 float output_value = gain * input_value;

 if   (min_value > output_value)
   return min_value;
 elsif (max_value < output_value)
   return max_value;
 else
   return output_value;
}

aqua_driver::init_aqua_driver(float displacement_gain,
                              float velocity_gain,
                              float current_feedback_gain)
{
 // set final gain to 1 and clip at +/- 75 VDC
 init_amplifier_clipper(1.0, 75.0, -75.0);

 theta_gain   = displacement_gain;
 omega_gain   = velocity_gain;
 current_gain = current_feedback_gain;
}

float
aqua_driver::drive(float theta_error, float omega, float current)
{
 return (amplify( theta_gain   * theta_error
                - omega_gain   * omega
                + current_gain * current));
}
```

```
// driver specs (theta, omega, current)
#define SHOULDER_DRIVER_GAINS  150.0, 5.0, 0.5
#define KNEE1_DRIVER_GAINS     150.0, 5.0, 0.5
#define KNEE2_DRIVER_GAINS     150.0, 5.0, 0.5

void makeJ1driver(aqua_driver& d)
{
 d.init_aqua_driver(SHOULDER_DRIVER_GAINS);
}

void makeJ2driver(aqua_driver& d);
{
 d.init_aqua_driver(KNEE1_DRIVER_GAINS);
}

void makeJ3driver(aqua_driver& d);
{
 d.init_aqua_driver(KNEE2_DRIVER_GAINS);
}
```

// file "joint_actuator.h"

```
#ifndef __JA_H__
#define __JA_H__

#include <>
#include ""

class aqua_joint_actuator {
protected:
 float          ordered_theta;
 amplifier_clipper da_converter;
 amplifier_clipper fv_converter;
 aqua_driver     d;
 motor           m;

public:
 aqua_joint_actuator() {}
 void reset(float theta);
 void input_order(float delta_theta);
 float torque(float current_theta, float current_omega);
};

class shoulder_actuator : public aqua_joint_actuator {

public:
 shoulder_actuator(float theta);
};
```

```
class knee1_actuator : public aqua_joint_actuator {

public:
 knee1_actuator(float theta);
 float torque(float current_theta, float current_omega);
};

class knee2_actuator : public aqua_joint_actuator {

public:
 knee2_actuator(float theta);
 float torque(float current_theta, float current_omega);
};

#endif
```

// file "joint_actuator.c"

```
#include "joint_actuator.h"

// aqua_joint_actuator (parent class) functions

void
aqua_joint_actuator::reset(float theta);
{
 ordered_theta = theta;
 m.Ia = 0.0;
}

void
aqua_joint_actuator::input_order(float delta_theta);
{
 ordered_theta += delta_theta;
}

float
aqua_joint_actuator::torque(float current_theta, float current_omega);
{
 float source_voltage;
 source_voltage = d->drive
            (da_converter->amplify(current_theta - ordered_theta),
             fv_converter->amplify(current_omega),
             m.Ia);

 return (m->developed_torque(source_voltage));
}
```

```
// derrived class specs

// output 10 volts for 6144 pulse count
#define DA_CONVERTER_RATIO 10.0/6144.0

// 100 pulses drive the motor 1 revolution
#define PULSES_PER_REV 100.0

// output 3 volts per 1000 RPMs
#define FV_CONVERTER_RATIO 3.0/1000.0

// harmonic gear only for shoulder
#define SHOULDER_REDUCTION 161.0

// harmonic and bevel gears for knees
#define KNEE1_REDUCTION 160.0*3.0
#define KNEE2_REDUCTION 160.0*2.0

// derrived class functions

shoulder_actuator::shoulder_actuator(float theta)
{
 ordered_theta = theta;
 da_converter->init_amplifier_clipper
  (SHOULDER_REDUCTION * PULSES_PER_REV * DA_CONVERTER_RATIO, 10.0, -10.0);
 fv_converter->init_amplifier_clipper
  (SHOULDER_REDUCTION * FV_CONVERTER_RATIO, 10.0, -10.0);
 makeJ1driver(&d);
 makeRA20(&m);
}

knee1_actuator::knee1_actuator(float theta)
{
 ordered_theta = theta;
 da_converter->init_amplifier_clipper
  (KNEE1_REDUCTION * PULSES_PER_REV * DA_CONVERTER_RATIO, 10.0, -10.0);
 fv_converter->init_amplifier_clipper
  (KNEE1_REDUCTION * FV_CONVERTER_RATIO, 10.0, -10.0);
 makeJ2driver(&d);
 makeRH25(&m);
}

float
knee1_actuator::torque(float current_theta, float current_omega)
{
 return (3.0 * aqua_joint_actuator::torque(current_theta, current_omega/3.0));
}
```

```
knee2_actuator::knee2_actuator(float theta)
{
 ordered_theta = theta;
 da_converter->init_amplifier_clipper
  (KNEE2_REDUCTION * PULSES_PER_REV * DA_CONVERTER_RATIO, 10.0, -10.0);
 fv_converter->init_amplifier_clipper
  (KNEE2_REDUCTION * FV_CONVERTER_RATIO, 10.0, -10.0);
 makeJ3driver(&d);
 makeRH25(&m);
}

float
knee2_actuator::torque(float current_theta, float current_omega)
{
 return (2.0 * aqua_joint_actuator::torque(current_theta, current_omega/2.0));
}
```

APPENDIX C

LOADING AND OPERATING INSTRUCTIONS

To run demo, start LISP Interpreter and call:

```
(load "aqua-loader")
```

This file loads the source code in the correct sequence and makes calls to run the demo.
Additional runs may be observed by calling:

```
(drop-aqua)
```

SOURCE CODE FILES

; "aqua-loader"

```
; LOADER FOR AQUA-ROBOT

; define loader/compiler functions
;
(load "load-files.cl")

; aqua-robot loader/compiler functions
;
; (load-aqua)
(load-compiled-aqua)
; (compile-and-load-aqua)

(defun drop-aqua ()
  (restart-aqua)
  (do () (nil)
    (dotimes (i *loops*) (update-aquarobot aqua-1))
    (new-picture)))

(aqua-picture)
(drop-aqua)
```

; "load-files.cl"

```
(defun load-aqua ()
  ; general purpose files
  (load "misc.cl")
  (load "vector.cl")
  (load "matrix.cl")
  (load "kinematics.cl")
  (load "rigid-body.cl")
  (load "strobe-camera.cl")
  (load "link.cl")
  ; aqua-robot specific files
  (load "aqua-data.cl")
  (load "aqua-link.cl")
  (load "aqua.cl")
  (load "aqua-leg.cl")
  (load "aqua-inv-kinematics.cl")
  (load "aqua-jacobian.cl")
  (load "aqua-update-forces-and-torques.cl"))

(defun load-compiled-aqua ()
  (load "misc.fasl")
  (load "vector.fasl")
  (load "matrix.fasl")
  (load "kinematics.fasl")
  (load "rigid-body.fasl")
  (load "strobe-camera.fasl")
  (load "link.fasl")
  (load "aqua-data.fasl")
  (load "aqua-link.fasl")
  (load "aqua.fasl")
  (load "aqua-leg.fasl")
  (load "aqua-inv-kinematics.fasl")
  (load "aqua-jacobian.fasl")
  (load "aqua-update-forces-and-torques.fasl"))
```

```
(defun compile-and-load-aqua ()
; (compile-file "misc.cl")
  (load "misc.fasl")
; (compile-file "vector.cl")
  (load "vector.fasl")
; (compile-file "matrix.cl")
  (load "matrix.fasl")
  (compile-file "kinematics.cl")
  (load "kinematics.fasl")
  (compile-file "rigid-body.cl")
  (load "rigid-body.fasl")
  (compile-file "strobe-camera.cl")
  (load "strobe-camera.fasl")
  (compile-file "link.cl")
  (load "link.fasl")
  (compile-file "aqua-data.cl")
  (load "aqua-data.fasl")
  (compile-file "aqua-link.cl")
  (load "aqua-link.fasl")
  (compile-file "aqua.cl")
  (load "aqua.fasl")
  (compile-file "aqua-leg.cl")
  (load "aqua-leg.fasl")
  (compile-file "aqua-inv-kinematics.cl")
  (load "aqua-inv-kinematics.fasl")
  (compile-file "aqua-jacobian.cl")
  (load "aqua-jacobian.fasl")
  (compile-file "aqua-update-forces-and-torques.cl")
  (load "aqua-update-forces-and-torques.fasl"))
```

; "misc.cl"

```
(defun atan2 (dx dy)
  (cond ((zerop dx) (cond ((zerop dy) 0.0)
                          ((< dy 0)   (- (* 0.5 pi)))
                          (t          (* 0.5 pi))))
        ((< dx 0)   (cond ((< dy 0)   (- (atan (/ dy dx)) pi))
                          (t          (+ (atan (/ dy dx)) pi))))
        (t          (atan (/ dy dx)))))

; returns angle in degrees.
(defun atan2d (dx dy) (rad-to-deg (atan2 dx dy)))

(defun sqr (x) (* x x))

(defconstant rad-to-deg-multiplier (/ 180 pi))
(defun rad-to-deg (rad) (* rad rad-to-deg-multiplier))

(defconstant deg-to-rad-multiplier (/ pi 180))
(defun deg-to-rad (deg) (* deg deg-to-rad-multiplier))

;Returns first n elements of list.
(defun ncar (n list)
  (cond ((zerop n) nil)
        (t (cons (car list) (ncar (1- n) (cdr list))))))
```

```
; "vector.cl"

; A vector is a list of numerical atoms.

(defun vector-add (vector-1 vector-2) (mapcar '+ vector-1 vector-2))

(defun vector-subtract (vector-1 vector-2) (mapcar '- vector-1 vector-2))

(defun scalar-multiply (scalar vector)
  (cond ((null vector) nil)
        (t (cons (* scalar (car vector))
                 (scalar-multiply scalar (cdr vector))))))

(defun dot-product (x y)
  (apply '+ (mapcar '* x y)))     ;A matrix is a list of row vectors.

(defun cross-product (x y)        ;x and y are 3D vectors.
  (list (- (* (cadr x)  (caddr y)) (* (caddr x) (cadr y)))
        (- (* (caddr x) (car y))   (* (car x)   (caddr y)))
        (- (* (car x)   (cadr y))  (* (cadr x)  (car y)))))

(defun vector-length (vector) (sqrt (dot-product vector vector)))

(defun distance-between (x y) ;points x and y represented by vectors.
  (vector-length (vector-subtract x y)))

; returns a vector (0*(one-position - 1)  1  0*(length-one-position))
(defun unit-vector (one-position length)
  (do ((n length (1- n))
       (vector nil (cons (cond ((= one-position n) 1) (t 0)) vector)))
      ((zerop n) vector)))

(defun append1 (L) (append L '(1)))
```

```
; "matrix.cl"


; requires VECTOR.CL
; requires MISC.CL "ncar"
;
; A matrix is a list of row vectors.

(defun transpose (A)
 (cond ((null (cdr A)) (mapcar 'list (car A)))
    (t (mapcar 'cons (car A) (transpose (cdr A))))))

(defun post-multiply (M x)   ;M is a square matrix. x is a conformable vector.
 (cond ((null (cdr M)) (list (dot-product (car M) x)))
    (t (cons (dot-product (car M) x) (post-multiply (cdr M) x)))))

(defun pre-multiply (vector matrix)
 (post-multiply (transpose matrix) vector))

(defun matrix-multiply (A B)    ;A and B are conformable matrices.
 (cond ((null (cdr A)) (list (pre-multiply (car A) B)))
    (t (cons (pre-multiply (car A) B) (matrix-multiply (cdr A) B)))))

(defun chain-multiply (L)       ;L is a list of names of conformable matrices.
 (cond ((null (cddr L)) (matrix-multiply (eval (car L)) (eval (cadr L))))
    (t (matrix-multiply (eval (car L)) (chain-multiply (cdr L))))))

(defun cycle-left (matrix) (mapcar 'row-cycle-left matrix))

(defun row-cycle-left (row) (append (cdr row) (list (car row))))

(defun cycle-up (matrix) (append (cdr matrix) (list (car matrix))))

(defun unit-matrix (size)
 (do ((row-number size (1- row-number))
    (I nil (cons (unit-vector row-number size) I)))
   ((zerop row-number) I)))

(defun concat-matrix (A B)  ;A and B are matrices with equal number of rows.
 (cond ((null A) B)
    (t (cons (append (car A) (car B)) (concat-matrix (cdr A) (cdr B))))))

(defun augment (matrix)
 (concat-matrix matrix (unit-matrix (length matrix))))

(defun normalize-row (row) (scalar-multiply (/ 1.0 (car row)) row))
```

```
(defun solve-first-column (matrix)      ;Reduces first column to (1 0 ... 0).
 (do* ((L1 matrix (cdr L1))
       (L2 (normalize-row (car matrix)))
       (L3 (list L2) (cons (vector-add (car L1)
                                       (scalar-multiply (- (caar L1)) L2)) L3)))
      ((null (cdr L1)) (reverse L3))))

(defun square-car (M) ;Returns square matrix extracted from front of matrix M.
 (do ((m (length M))
      (L1 M (cdr L1))
      (L2 nil (cons (ncar m (car L1)) L2)))
     ((null L1) (reverse L2))))

;;L is a list of lists. This function finds list with
;largest car and moves it to head of list of lists.
(defun max-car-first (L)
 (cond ((null (cdr L)) L)
       (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L))))) L
              (append (max-car-first (cdr L)) (list (car L)))))))

;Applies max-car-first to first n elements of list.
(defun nmax-car-first (n list)
 (append (max-car-first (ncar n list)) (nthcdr n list)))

(defun matrix-inverse (M)
 (do ((M1 (max-car-first (augment M))
          (cond ((null M1) nil)
                (t (nmax-car-first n (cycle-left (cycle-up M1))))))
      (n (1- (length M)) (1- n)))
     ((or (minusp n) (null M1)) (cond ((null M1) nil) (t (square-car M1))))
     (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))
```

; "kinematics.cl"

```
; requires MATRIX.CL

(defun dh-matrix (cosrotate sinrotate costwist sintwist length translate)
 (list (list cosrotate (- (* costwist sinrotate))
        (* sintwist sinrotate) (* length cosrotate))
    (list sinrotate (* costwist cosrotate)
        (- (* sintwist cosrotate)) (* length sinrotate))
    (list 0. sintwist costwist translate) (list 0. 0. 0. 1.)))

(defun mdh-matrix (cosrotate sinrotate
              costwist-i-1 sintwist-i-1
              length-i-1 translate)
 (list (list cosrotate (- sinrotate) 0. length-i-1)
     (list (* sinrotate costwist-i-1) (* cosrotate costwist-i-1)
        (- sintwist-i-1) (- (* sintwist-i-1 translate)))
     (list (* sinrotate sintwist-i-1) (* cosrotate sintwist-i-1)
        costwist-i-1 (* costwist-i-1 translate))
     (list 0. 0. 0. 1.)))

(defun homogeneous-transform (azimuth elevation roll x y z)
 (rotation-and-translation (sin azimuth) (cos azimuth) (sin elevation)
 (cos elevation) (sin roll) (cos roll) x y z))

(defun rotation-and-translation (spsi cpsi sth cth sphi cphi x y z)
 (list (list (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
        (+ (* cpsi sth cphi) (* spsi sphi)) x)
    (list (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
        (- (* spsi sth cphi) (* cpsi sphi)) y)
    (list (- sth) (* cth sphi) (* cth cphi) z)
    (list 0. 0. 0. 1.)))

(defun inverse-H (H)
 (let* ((minus-P (list (- (fourth (first H)))
                (- (fourth (second H)))
                (- (fourth (third H)))))
      (inverse-R (transpose (square-car (reverse (rest (reverse H))))))
      (inverse-P (post-multiply inverse-R minus-P)))
     (append (concat-matrix inverse-R (transpose (list inverse-P)))
          (list (list 0 0 0 1)))))
```

```
; "rigid-body.cl"

; requires KINEMATICS.CL

(defclass rigid-body  ()
 ((location          ;The three-vector (x y z) in world coordinates.
   :initarg :location
   :accessor location)
  (velocity          ;The six-vector (u v w p q r) in body coordinates.
   :initarg :velocity
   :accessor velocity)
  (acceleration      ;The vector (u-dot v-dot w-dot p-dot q-dot r-dot).
   :accessor acceleration)
  (forces-and-torques  ;The vector (Fx Fy Fz L M N) in body coordinates.
   :accessor forces-and-torques)
  (moments-of-inertia  ;The vector (Ix Iy Iz) in principal axis coordinates.
   :initarg :moments-of-inertia
   :accessor moments-of-inertia)
  (mass
   :initarg :mass
   :accessor mass)
  (node-list         ;List of vertices for wire frame model
   :initarg :node-list
   :accessor node-list)
  (polygon-list      ;Sets of above vertices defining polygons
   :initarg :polygon-list ;Ex:'((1 2 3)(2 3 4 5)(4 5 6))
   :accessor polygon-list)
  (transformed-node-list
   :accessor transformed-node-list)
  (H-matrix
   :accessor H-matrix)
  (current-time
   :accessor current-time)))

(defmethod move ((body rigid-body) azimuth elevation roll x y z)
 (setf (H-matrix body)
     (homogeneous-transform azimuth elevation roll x y z))
 (transform-node-list body)
 (update-position body))
```

```
(defmethod move-incremental ((body rigid-body) increment-list)
 (setf (H-matrix body)
     (matrix-multiply (H-matrix body) (homogeneous-transform
                          (first increment-list)
                          (second increment-list)
                          (third increment-list)
                          (fourth increment-list)
                          (fifth increment-list)
                          (sixth increment-list))))
 (transform-node-list body)
 (update-position body))


(defmethod get-delta-t ((body rigid-body))
 (let* ((new-time (get-internal-real-time))
     (delta-t (/ (- new-time (current-time body)) 1000)))
  (setf (current-time body) new-time)
  delta-t))

(defmethod start-timer ((body rigid-body))
 (setf (current-time body) (get-internal-real-time)))

(defmethod update-rigid-body ((body rigid-body))    ;Euler integration.
 (let* ((delta-t (get-delta-t body)))
  (update-H-matrix body delta-t)
  (transform-node-list body)
  (update-position body)
  (update-velocity body delta-t)
  (update-acceleration body)))

(defmethod update-acceleration ((body rigid-body))
 (setf (acceleration body) ;(list u-dot v-dot w-dot p-dot q-dot r-dot)
  (multiple-value-bind                ;Assumes principal axis
    (Fx Fy Fz L M N  u v w p q r  Ix Iy Iz) ;coordinates with origin at
    (values-list                      ;center of gravity of body.
     (append
      (forces-and-torques body) (velocity body) (moments-of-inertia body)))
    (list (+ (* v r) (* -1 w q) (/ Fx (mass body))
         (* *gravity* (first (third (H-matrix body)))))
       (+ (* w p) (* -1 u r) (/ Fy (mass body))
         (* *gravity* (second (third (H-matrix body)))))
       (+ (* u q) (* -1 v p) (/ Fz (mass body))
         (* *gravity* (third (third (H-matrix body)))))
    (/ (+ (* (- Iy Iz) q r) L) Ix)
    (/ (+ (* (- Iz Ix) r p) M) Iy)
    (/ (+ (* (- Ix Iy) p q) N) Iz)))))
```

```
(defmethod update-velocity ((body rigid-body) delta-t)
 (setf (velocity body)
  (vector-add (velocity body)
       (scalar-multiply delta-t (acceleration body)))))

(defmethod update-H-matrix ((body rigid-body) delta-t)
 (setf (H-matrix body)
  (matrix-multiply
   (H-matrix body) (homogeneous-transform
            (* delta-t (sixth (velocity body)))
            (* delta-t (fifth (velocity body)))
            (* delta-t (fourth (velocity body)))
            (* delta-t (first (velocity body)))
            (* delta-t (second (velocity body)))
            (* delta-t (third (velocity body)))))))

(defmethod transform-node-list ((body rigid-body))
 (setf (transformed-node-list body)
  (mapcar #'(lambda (node-location)
       (post-multiply (H-matrix body) node-location))
      (node-list body))))

(defmethod update-position ((body rigid-body))
 (setf (location body) (ncar 3 (first (transformed-node-list body)))))

; (defconstant *gravity* 32.2185)

(defmethod world-to-body ((body rigid-body) xyz-pos)
  (ncar 3 (post-multiply (inverse-H (H-matrix body))
                (append xyz-pos '(1)))))

(defmethod body-to-world ((body rigid-body) xyz-pos)
  (ncar 3 (post-multiply (H-matrix body) (append xyz-pos '(1)))))
```

```
; "strobe-camera.cl"

; requires RIGID-BODY.CL

(require :xcw)
(cw:initialize-common-windows)

(defclass strobe-camera (rigid-body)
 ((focal-length
   :accessor focal-length
   :initform 6)
  (camera-window
   :accessor camera-window
   :initform (cw:make-window-stream :borders 5
                                    :left 500
                                   :bottom 500
                                  :width 300
                                  :height 300
                                  :title "strobe-camera-image"
                                  :activate-p t))
  (H-matrix
   :initform (homogeneous-transform .3 -.3 0 -300 -90 -90))
  (inverse-H-matrix
   :accessor inverse-H-matrix
   :initform (inverse-H (homogeneous-transform .3 -.3 0 -300 -90 -90)))
  (enlargement-factor
   :accessor enlargement-factor
   :initform 30)))

(defmethod erase-camera-window ((camera strobe-camera))
 (cw:clear (camera-window camera)))

(defmethod move ((camera strobe-camera) azimuth elevation roll x y z)
 (setf (H-matrix camera) (homogeneous-transform azimuth elevation roll x y z))
 (setf (inverse-H-matrix camera) (inverse-H (H-matrix camera))))

(defmethod take-picture ((camera strobe-camera) (body rigid-body))
 (let ((camera-space-node-list (mapcar #'(lambda (node-location)
      (post-multiply (inverse-H-matrix camera) node-location))
            (transformed-node-list body))))
  (dolist (polygon (polygon-list body))
   (clip-and-draw-polygon camera polygon camera-space-node-list))))
```

```
(defmethod clip-and-draw-polygon
 ((camera strobe-camera) polygon node-coord-list)
 (do* ((initial-point (nth (first polygon) node-coord-list))
       (from-point initial-point to-point)
       (remaining-nodes (rest polygon) (rest remaining-nodes))
       (to-point (nth (first remaining-nodes) node-coord-list)
                 (if (not (null (first remaining-nodes)))
                     (nth (first remaining-nodes) node-coord-list))))
      ((null to-point)
       (draw-clipped-projection camera from-point initial-point))
      (draw-clipped-projection camera from-point to-point)))

(defmethod draw-clipped-projection ((camera strobe-camera) from-point to-point)
 (cond ((and (<= (first from-point) (focal-length camera))
             (<= (first to-point) (focal-length camera))) nil)
       ((<= (first from-point) (focal-length camera))
        (draw-line-in-camera-window camera
         (perspective-transform camera (from-clip camera from-point to-point))
         (perspective-transform camera to-point)))
       ((<= (first to-point) (focal-length camera))
        (draw-line-in-camera-window camera
         (perspective-transform camera from-point)
         (perspective-transform camera (to-clip camera from-point to-point))))
       (t (draw-line-in-camera-window camera
         (perspective-transform camera from-point)
         (perspective-transform camera to-point)))))

(defmethod from-clip ((camera strobe-camera) from-point to-point)
 (let ((scale-factor (/ (- (focal-length camera) (first from-point))
                        (- (first to-point) (first from-point)))))
      (list (+ (first from-point)
               (* scale-factor (- (first to-point) (first from-point))))
            (+ (second from-point)
               (* scale-factor (- (second to-point) (second from-point))))
            (+ (third from-point)
               (* scale-factor (- (third to-point) (third from-point)))) 1)))

(defmethod to-clip ((camera strobe-camera) from-point to-point)
 (from-clip camera to-point from-point))

(defmethod draw-line-in-camera-window ((camera strobe-camera) start end)
 (cw:draw-line (camera-window camera)
               (cw:make-position :x (first start) :y (second start))
               (cw:make-position :x (first end) :y (second end))
               :brush-width 0))
```

```
(defmethod perspective-transform ((camera strobe-camera) point-in-camera-space)
 (let* ((enlargement-factor (enlargement-factor camera))
    (focal-length (focal-length camera))
    (x (first point-in-camera-space))  ;x axis is along optical axis
    (y (second point-in-camera-space)) ;y is out right side of camera
    (z (third point-in-camera-space))) ;z is out bottom of camera
   (list (+ (round (* enlargement-factor (/ (* focal-length y) x)))
        150)                ;to right in camera window
       (+ 150 (round (* enlargement-factor (/ (* focal-length (- z)) x))
                  )))))  ;up in camera window
```

```
; "link.cl"

; requires RIGID-BODY.CL

(defclass link (rigid-body)
  ((motion-limit-flag
    :initform nil
    :accessor motion-limit-flag)
   (twist-angle
    :initarg :twist-angle
    :accessor twist-angle)
   (link-length
    :initarg :link-length
    :accessor link-length)
   (inboard-joint-angle
    :initarg :inboard-joint-angle
    :accessor inboard-joint-angle)
   (inboard-joint-displacement
    :initarg :inboard-joint-displacement
    :accessor inboard-joint-displacement)
   (inboard-link
    :initarg :inboard-link
    :accessor inboard-link)
   (A-matrix
    :accessor A-matrix)
; added for mdh
   (twist-angle-i-1
    :initarg :twist-angle-i-1
    :accessor twist-angle-i-1)
   (link-length-i-1
    :initarg :link-length-i-1
    :accessor link-length-i-1)
   (T-matrix
    :accessor T-matrix)))

(defclass rotary-link (link)
  ((min-joint-angle
    :initarg :min-joint-angle
    :accessor min-joint-angle)
   (max-joint-angle
    :initarg :max-joint-angle
    :accessor max-joint-angle)))
```

```
(defclass sliding-link (link)
  ((min-joint-displacement
    :initarg :min-joint-displacement
    :accessor min-joint-displacement)
   (max-joint-displacement
    :initarg :max-joint-displacement
    :accessor max-joint-displacement)))
```

```
; "aqua-data.cl"

; load after MISC.CL

(defconstant *dt* 0.01)  ; delta-t each update.
(defconstant *loops* 2)  ; updates between draws.

(defconstant link0length 37.5)
(defconstant link1length 20.0)
(defconstant link2length 52.0)
(defconstant link3length 102.0)
(defconstant flag-length 25.0)

; leg attachment angles.
(defconstant leg1-angle (deg-to-rad 0))
(defconstant leg2-angle (deg-to-rad 60))
(defconstant leg3-angle (deg-to-rad 120))
(defconstant leg4-angle (deg-to-rad 180))
(defconstant leg5-angle (deg-to-rad 240))
(defconstant leg6-angle (deg-to-rad 300))

; initial position and orientation in world coordinates.
(defconstant azimuth-init    (deg-to-rad 0.0))
(defconstant elevation-init  (deg-to-rad 0.0))
(defconstant roll-init       (deg-to-rad 9.0))
(defconstant x-init          0.0)
(defconstant y-init          0.0)
(defconstant z-init         -135.0)

; initial (default) joint angles.
(defconstant joint1-init (deg-to-rad   0.0))
(defconstant joint2-init (deg-to-rad   25.0))
(defconstant joint3-init (deg-to-rad -115.0))
(defconstant default-angles (list joint1-init joint2-init joint3-init))

; joint spring constants. (fill in : Kg-cm2/sec2 per radian)
(defconstant joint1-K -2000000) ; 5000000 = Scott's 500 Nm per radian.
(defconstant joint2-K -2000000)
(defconstant joint3-K -2000000)
(defconstant spring-constants (list joint1-K joint2-K joint3-K))
; joint spring damping constants. (fill in : Kg-cm2/sec2 per radian/sec)
(defconstant joint1-D -800000)  ; 800000 = Scott's 80 Nm-sec per radian/sec.
(defconstant joint2-D -800000)
(defconstant joint3-D -800000)
(defconstant spring-damping-constants (list joint1-D joint2-D joint3-D))
```

```
; joint limits.
;(defconstant joint1-min-limit (deg-to-rad  -60.0))
;(defconstant joint1-max-limit (deg-to-rad   60.0))
;(defconstant joint2-min-limit (deg-to-rad -106.6))
;(defconstant joint2-max-limit (deg-to-rad   73.4))
;(defconstant joint3-min-limit (deg-to-rad -156.4))
;(defconstant joint3-max-limit (deg-to-rad   23.6))
(defconstant joint1-min-limit -50.0)
(defconstant joint1-max-limit  50.0)
(defconstant joint2-min-limit -50.0)
(defconstant joint2-max-limit  50.0)
(defconstant joint3-min-limit -50.0)
(defconstant joint3-max-limit  50.0)

; mass in Kg.
(defconstant aqua-body-mass 500.0)
;(defconstant link1mass 0.0)
;(defconstant link2mass 0.0)
;(defconstant link3mass 0.0)

; (Ix Iy Iz)-Kg-cm.  in principal axis coordinates.
; assumes solid cylindrical body of constant density.
(defconstant aqua-body-height 50.0)
(defconstant aqua-body-radius 30.0)
(defconstant aqua-body-Ix
        (+ (* (/ 1 4) aqua-body-mass (sqr aqua-body-radius))
          (* (/ 1 12) aqua-body-mass (sqr aqua-body-height))))
(defconstant aqua-body-Iy aqua-body-Ix)
(defconstant aqua-body-Iz (* (/ 1 2) aqua-body-mass (sqr aqua-body-radius)))
(defconstant aqua-body-inertia (list aqua-body-Ix aqua-body-Iy aqua-body-Iz))

; center of mass.
(defconstant body-mass-center '(0 0 0))
;(defconstant link1mass-center (list (/ link1length 2) 0 0))
;(defconstant link2mass-center (list (/ link2length 2) 0 0))
;(defconstant link3mass-center (list (/ link3length 2) 0 0))

(defconstant *gravity* 980.0) ;cm/sec/sec.
```

; "aqua-link.cl"

```
; requires LINK.CL
; requires AQUA-DATA.CL

(defclass link0 (rotary-link)
  ((twist-angle :initform 0)
   (link-length :initform link0length)
   (inboard-joint-angle :initform 0)
   (inboard-joint-displacement :initform 0)
   (min-joint-angle :initform (deg-to-rad -360))
   (max-joint-angle :initform (deg-to-rad 360))
   (node-list :initform (list (list 0 0 0 1) (list 0 0 0 1)
                              (list link0length 0 0 1)))    ; for mdh
;                             (list (- link0length) 0 0 1))) ; for dh
   (polygon-list :initform '((1 2)))))

(defclass link1 (rotary-link)
  ((twist-angle :initform (deg-to-rad -90))
   (link-length :initform link1length)
   (inboard-joint-angle :initform joint1-init)
   (inboard-joint-displacement :initform 0)
   (min-joint-angle :initform joint1-min-limit)
   (max-joint-angle :initform joint1-max-limit)
   (node-list :initform (list (list 0 0 0 1) (list 0 0 0 1)
                              (list link1length 0 0 1)))    ; for mdh
;                             (list (- link1length) 0 0 1))) ; for dh
   (polygon-list :initform '((1 2)))))

(defclass link2 (rotary-link)
  ((twist-angle :initform 0)
   (link-length :initform link2length)
   (inboard-joint-angle :initform joint2-init)
   (inboard-joint-displacement :initform 0)
   (min-joint-angle :initform joint2-min-limit)
   (max-joint-angle :initform joint2-max-limit)
   (node-list :initform (list (list 0 0 0 1) (list 0 0 0 1)
                              (list link2length 0 0 1)))    ; for mdh
;                             (list (- link2length) 0 0 1))) ; for dh
   (polygon-list :initform '((1 2)))))
```

```
(defclass link3 (rotary-link)
  ((twist-angle :initform 0)
   (link-length :initform link3length)
   (inboard-joint-angle :initform joint3-init)
   (inboard-joint-displacement :initform 0)
   (min-joint-angle :initform joint3-min-limit)
   (max-joint-angle :initform joint3-max-limit)
   (node-list :initform (list (list 0 0 0 1) (list 0 0 0 1)
                              (list link3length 0 0 1)))    ; for mdh
;                              (list (- link3length) 0 0 1))) ; for dh
   (polygon-list :initform '((1 2)))))

;; for dh
(defmethod update-A-matrix ((link link))
  (with-slots (twist-angle link-length inboard-joint-angle
    inboard-joint-displacement A-matrix) link
    (setf A-matrix
        (dh-matrix (cos inboard-joint-angle) (sin inboard-joint-angle)
                (cos twist-angle)     (sin twist-angle)
                link-length        inboard-joint-displacement))))

; added for mdh
(defmethod update-T-matrix ((link link))
  (with-slots (twist-angle-i-1 link-length-i-1 inboard-joint-angle
    inboard-joint-displacement T-matrix) link
    (setf T-matrix
        (mdh-matrix (cos inboard-joint-angle) (sin inboard-joint-angle)
                (cos twist-angle-i-1)  (sin twist-angle-i-1)
                link-length-i-1     inboard-joint-displacement))))

(defmethod rotate ((link link) angle)
  (setf (inboard-joint-angle link) angle)
  (update-T-matrix link)
  (setf (H-matrix link) (matrix-multiply (H-matrix (inboard-link link))
                                         (T-matrix link)))
  (transform-node-list link))

(defmethod rotate-link ((link link) angle)
  (cond ((> angle (max-joint-angle link))
         (rotate link (max-joint-angle link))
         (setf (motion-limit-flag link) t))
        ((< angle (min-joint-angle link))
         (rotate link (min-joint-angle link))
         (setf (motion-limit-flag link) t))
        (t (rotate link angle) (setf (motion-limit-flag link) nil))))
```

```
; "aqua.cl"

; requires STROBE-CAMERA.CL
; requires AQUA-LEG.CL

(defclass aquarobot-body (rigid-body)
  ((mass :initform aqua-body-mass)
   (moments-of-inertia :initform aqua-body-inertia)
   (node-list
    :initform  (list (list 0 0 0 1)
                     (list (* (cos leg1-angle) link0length)
                         (* (sin leg1-angle) link0length) 0 1)
                     (list (* (cos leg2-angle) link0length)
                         (* (sin leg2-angle) link0length) 0 1)
                     (list (* (cos leg3-angle) link0length)
                         (* (sin leg3-angle) link0length) 0 1)
                     (list (* (cos leg4-angle) link0length)
                         (* (sin leg4-angle) link0length) 0 1)
                     (list (* (cos leg5-angle) link0length)
                         (* (sin leg5-angle) link0length) 0 1)
                     (list (* (cos leg6-angle) link0length)
                         (* (sin leg6-angle) link0length) 0 1)
                     (list link0length 0 (- flag-length) 1)))
   (polygon-list
    :initform '((1 2 3 4 5 6) (1 7)))))

(defclass aquarobot ()
 ((body
   :initform (make-instance 'aquarobot-body)
   :accessor body)
  (leg1
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg1-angle)
   :accessor leg1)
  (leg2
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg2-angle)
   :accessor leg2)
  (leg3
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg3-angle)
   :accessor leg3)
  (leg4
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg4-angle)
   :accessor leg4)
  (leg5
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg5-angle)
   :accessor leg5)
  (leg6
   :initform (make-instance 'aqua-leg :leg-attachment-angle leg6-angle)
   :accessor leg6)))
```

```
(defmethod world-to-aqua ((aqua aquarobot) xyz-pos)
  (world-to-body (body aqua) xyz-pos))

(defmethod aqua-to-world ((aqua aquarobot) xyz-pos)
  (body-to-world (body aqua) xyz-pos))

;(defmethod initialize ((aqua aquarobot))
  (setf (H-matrix (body aqua))
      (homogeneous-transform azimuth-init elevation-init roll-init
                  x-init y-init z-init))
  (transform-node-list (body aqua))
  (update-position (body aqua))
  (setf (forces-and-torques (body aqua)) '(0 0 0 0 0 0))
  (setf (acceleration (body aqua)) '(0 0 0 0 0 0))
  (setf (velocity (body aqua)) '(0 0 0 0 0 0))
  (start-timer (body aqua))
  (initialize-leg (leg1 aqua) (body aqua))
  (initialize-leg (leg2 aqua) (body aqua))
  (initialize-leg (leg3 aqua) (body aqua))
  (initialize-leg (leg4 aqua) (body aqua))
  (initialize-leg (leg5 aqua) (body aqua))
  (initialize-leg (leg6 aqua) (body aqua)))

(defun aqua-picture ()
  (setf aqua-1 (make-instance 'aquarobot))
  (initialize aqua-1)
  (move-incremental aqua-1 null-move-list);sets "prev-foot-pos".
  (setf camera-1 (make-instance 'strobe-camera))
  (take-picture camera-1 aqua-1))

(defmethod take-picture ((camera strobe-camera) (aqua aquarobot))
  (take-picture camera (body aqua))
  (take-picture camera (leg1 aqua))
  (take-picture camera (leg2 aqua))
  (take-picture camera (leg3 aqua))
  (take-picture camera (leg4 aqua))
  (take-picture camera (leg5 aqua))
  (take-picture camera (leg6 aqua)))

(defun new-picture ()
  (erase-camera-window camera-1)
  (take-picture camera-1 aqua-1))
```

```
(defmethod move-incremental ((aqua aquarobot) increment-list)
  (move-incremental (body aqua) (first increment-list))
  (move-incremental (leg1 aqua) (second increment-list))
  (move-incremental (leg2 aqua) (third increment-list))
  (move-incremental (leg3 aqua) (fourth increment-list))
  (move-incremental (leg4 aqua) (fifth increment-list))
  (move-incremental (leg5 aqua) (sixth increment-list))
  (move-incremental (leg6 aqua) (seventh increment-list)))

(defconstant null-move-list '((0 0 0 0 0 0) (0 0 0) (0 0 0) (0 0 0)
 (0 0 0) (0 0 0) (0 0 0)))

(defmethod feasible-movep ((aqua aquarobot) allowable-sinkage
                           allowable-slippage)
(and (feasible-movep (leg1 aqua) allowable-sinkage allowable-slippage)
     (feasible-movep (leg2 aqua) allowable-sinkage allowable-slippage)
     (feasible-movep (leg3 aqua) allowable-sinkage allowable-slippage)
     (feasible-movep (leg4 aqua) allowable-sinkage allowable-slippage)
     (feasible-movep (leg5 aqua) allowable-sinkage allowable-slippage)
     (feasible-movep (leg6 aqua) allowable-sinkage allowable-slippage)))

(defun restart-aqua ()
  (initialize aqua-1)
  (move-incremental aqua-1 null-move-list);sets "prev-foot-pos".
  (new-picture))

;replace some rigid-body functions:

(defmethod start-timer ((body aquarobot-body))
  (setf (current-time body) 0))

(defmethod get-delta-t ((body aquarobot-body))
  (let* ((delta-t *dt*)
         (new-time (+ (current-time body) delta-t)))
    (setf (current-time body) new-time)
    delta-t))

(defmethod update-aquarobot ((aqua aquarobot)) ;Euler integration.
  (let* ((body (body aqua))
         (delta-t (get-delta-t body)))
    (update-acceleration body)
    (update-velocity body delta-t)
    (update-H-matrix body delta-t)
    (transform-node-list body)
    (update-position body)
    (update-forces-and-torques aqua))) ;updates positions of legs
```

```
; "aqua-leg.cl"

; requires AQUA.CL
; requires AQUA-LINK.CL
; requires STROBE-CAMERA.CL

(defclass aqua-leg ()
 ((leg-attachment-angle
   :initarg :leg-attachment-angle
   :accessor leg-attachment-angle)
  (link0
   :initform (make-instance 'link0)
   :accessor link0)
  (link1
   :initform (make-instance 'link1)
   :accessor link1)
  (link2
   :initform (make-instance 'link2)
   :accessor link2)
  (link3
   :initform (make-instance 'link3)
   :accessor link3)
  (motion-complete-flag
   :initform nil
   :accessor motion-complete-flag)
  (previous-foot-position
   :initform nil
   :accessor previous-foot-position)
  (current-foot-position
   :initform nil
   :accessor current-foot-position)
  (foot-contact
   :initform nil
   :accessor foot-contact)))
```

```
(defmethod initialize-leg ((leg aqua-leg) (body aquarobot-body))
 (setf (foot-contact leg) nil)
 (setf (inboard-link (link0 leg)) body)
 (setf (inboard-link (link1 leg)) (link0 leg))
 (setf (inboard-link (link2 leg)) (link1 leg))
 (setf (inboard-link (link3 leg)) (link2 leg))
 ; added for mdh
 (setf (twist-angle-i-1 (link0 leg)) 0)
 (setf (twist-angle-i-1 (link1 leg)) (twist-angle (link0 leg)))
 (setf (twist-angle-i-1 (link2 leg)) (twist-angle (link1 leg)))
 (setf (twist-angle-i-1 (link3 leg)) (twist-angle (link2 leg)))
 (setf (link-length-i-1 (link0 leg)) 0)
 (setf (link-length-i-1 (link1 leg)) (link-length (link0 leg)))
 (setf (link-length-i-1 (link2 leg)) (link-length (link1 leg)))
 (setf (link-length-i-1 (link3 leg)) (link-length (link2 leg)))
 (set-default-angles leg))

(defmethod set-default-angles ((leg aqua-leg))
 (rotate-link (link0 leg) (leg-attachment-angle leg))
 (rotate-link (link1 leg) joint1-init)
 (rotate-link (link2 leg) joint2-init)
 (rotate-link (link3 leg) joint3-init)
 (setf (previous-foot-position leg) nil)
 (setf (current-foot-position leg)
     (ncar 3 (third (transformed-node-list (link3 leg))))) ; for mdh

(defmethod set-angles ((leg aqua-leg) angle-list)
  (rotate-link (link0 leg) (leg-attachment-angle leg))
  (rotate-link (link1 leg) (car angle-list))
  (rotate-link (link2 leg) (cadr angle-list))
  (rotate-link (link3 leg) (caddr angle-list)))

(defmethod take-picture ((camera strobe-camera) (leg aqua-leg))
 (take-picture camera (link1 leg))
 (take-picture camera (link2 leg))
 (take-picture camera (link3 leg)))
```

```
(defmethod move-incremental ((leg aqua-leg) increment-list)
 (rotate-link (link0 leg) (leg-attachment-angle leg))
 (rotate-link (link1 leg)
        (+ (first increment-list) (inboard-joint-angle (link1 leg))))
 (rotate-link (link2 leg)
        (+ (second increment-list) (inboard-joint-angle (link2 leg))))
 (rotate-link (link3 leg)
        (+ (third increment-list) (inboard-joint-angle (link3 leg))))
 (setf (previous-foot-position leg) (current-foot-position leg))
 (setf (current-foot-position leg)
     (ncar 3 (third (transformed-node-list (link3 leg))))) ; for mdh
;    (ncar 3 (first (transformed-node-list (link3 leg))))) ; for dh
 (setf (motion-complete-flag leg) (not (or (motion-limit-flag (link1 leg))
      (motion-limit-flag (link2 leg)) (motion-limit-flag (link3 leg))))))

(defmethod feasible-movep ((leg aqua-leg) allowable-sinkage allowable-slippage)
 (and (<= (third (current-foot-position leg)) allowable-sinkage)
    (or (minusp (third (current-foot-position leg)))
      (minusp (third (previous-foot-position leg)))
      (<= (vector-length (vector-slippage leg)) allowable-slippage))))

(defmethod vector-slippage ((leg aqua-leg))
 (vector-subtract (rest (reverse (previous-foot-position leg)))
           (rest (reverse (current-foot-position leg)))))

(defmethod current-joint-angles ((leg aqua-leg))
 (list (inboard-joint-angle (link1 leg))
     (inboard-joint-angle (link2 leg))
     (inboard-joint-angle (link3 leg))))
```

```
; "aqua-inverse-kinematics.cl"


; load after AQUA-LEG.CL
; load after AQUA-DATA.CL

(defconstant L2sqr (sqr link2length))
(defconstant L3sqr (sqr link3length))

; assumptions: dh coord system for link0 of respective leg:
;              origin at joint1,
;              x-axis directed away from center of body,
;              z-axis aligned with body z-axis;
;            foot-position = '(x y z).
(defun theta1 (foot-position)
  (if (< (car foot-position) 0)
      (atan2 (- (car foot-position)) (- (cadr foot-position)))
      (atan2 (car foot-position) (cadr foot-position))))

; assumptions: dh coord system for link1 of respective leg:
;              origin at joint2,
;              x-axis directed away from joint1,
;              z-axis aligned with body z-axis;
;            foot-position = '(x y z);
;            hyp = distance from joint2 to foot.
(defun theta2 (foot-position hyp hyp-sqr)
  (- (acos (/ (+ L2sqr hyp-sqr (- L3sqr)) (* 2 link2length hyp)))
     (if (< (car foot-position) 0)
         (- pi (asin (/ (caddr foot-position) hyp)))
         (asin (/ (caddr foot-position) hyp)))))

; assumptions: same as for theta2.
(defun theta3 (foot-position hyp-sqr)
  (- (acos (/ (+ L2sqr L3sqr (- hyp-sqr)) (* 2 link2length link3length))) pi))


; returns foot position with respect to joint 1 in link0 coord.
(defmethod foot-joint1/link0coord ((leg aqua-leg) foot-pos)
  (vector-subtract (world-to-body (link0 leg) foot-pos)
                   (list link0length 0 0)))

; returns foot position with respect to joint 2 in link1 coord.
; given foot-joint1/link0coord.
(defun foot-joint2/link1coord (foot-pos)
  (list (- (sqrt (+ (sqr (car foot-pos)) (sqr (cadr foot-pos)))) link1length)
        0 (caddr foot-pos)))
```

```
; returns list of joint angles required for given (world coord) foot position.
(defmethod aqua-inv-kin ((leg aqua-leg) foot-position)
  (let* ((pos0 (foot-joint1/link0coord leg foot-position))
         (pos1 (foot-joint2/link1coord pos0))
         (hyp-sqr (+ (sqr (car pos1)) (sqr (caddr pos1))))
         (hyp (sqrt hyp-sqr)))
        (list (theta1 pos0)
              (theta2 pos1 hyp hyp-sqr)
              (theta3 pos1 hyp-sqr))))
```

; "aqua-jacobian.cl"

```
(defmethod jacobian ((leg aqua-leg))
  (let* ((T01 (+ (leg-attachment-angle leg)
                 (inboard-joint-angle (link1 leg))))
         (S01 (sin T01))  (C01 (cos T01))
         (T2  (inboard-joint-angle (link2 leg)))
         (S2  (sin T2))   (C2 (cos T2))
         (T23 (+ T2 (inboard-joint-angle (link3 leg))))
         (S23 (sin T23))  (C23 (cos T23))
         (L1 link1length) (L2 link2length) (L3 link3length))
        (list (list (- (* (+ L1 (* L2 C2) (* L3 C23)) S01))
                    (- (* (+ (* L2 S2) (* L3 S23)) C01))
                    (- (* L3 C01 S23)))
              (list (* (+ L1 (* L2 C2) (* L3 C23)) C01)
                    (- (* (+ (* L2 S2) (* L3 S23)) S01))
                    (- (* L3 S01 S23)))
              (list 0
                    (- (+ (* L2 C2) (* L3 C23)))
                    (- (* L3 C23))))))

(defmethod inverse-jacobian ((leg aqua-leg))
  (matrix-inverse (jacobian leg)))

(defmethod foot-to-joint-rates ((leg aqua-leg) dX dY dZ)
  (post-multiply (inverse-jacobian leg) (list dX dY dZ)))

(defmethod joint-to-foot-rates ((leg aqua-leg) dtheta1 dtheta2 dtheta3)
  (post-multiply (jacobian leg) (list dtheta1 dtheta2 dtheta3)))
```

```
; "aquarobot-update-forces-and-torques.cl"


; load after AQUA-DATA.CL
; load after AQUA.CL
; load after AQUA-LEG.CL

(defmethod update-forces-and-torques ((aqua aquarobot))
  (setf (forces-and-torques (body aqua)) '(0 0 0 0 0 0)) ;clear last cycle.
  (add-leg-forces-and-torques (leg1 aqua))
  (add-leg-forces-and-torques (leg2 aqua))
  (add-leg-forces-and-torques (leg3 aqua))
  (add-leg-forces-and-torques (leg4 aqua))
  (add-leg-forces-and-torques (leg5 aqua))
  (add-leg-forces-and-torques (leg6 aqua)))

(defmethod add-leg-forces-and-torques ((leg aqua-leg))
 (if (or (foot-contact leg) (new-contact leg))
    (let* ((body (inboard-link (link0 leg)))
          (joint-angles (aqua-inv-kin leg (current-foot-position leg))))
         (set-angles leg joint-angles)
         (let* ((r (world-to-body body (current-foot-position leg)))
              (omega (cdddr (velocity body)))
              (foot-velocity    ; in body coordinates
                (vector-add
                  (scalar-multiply -1 (ncar 3 (velocity body)))
                  (cross-product r omega)))
              (torques (vector-add
                (mapcar '* spring-constants
                  (vector-subtract joint-angles default-angles))
                (mapcar '* spring-damping-constants
                  (post-multiply (inverse-jacobian leg) foot-velocity))))
              (resultant-force
               (scalar-multiply
                -1 (post-multiply
                     (matrix-inverse (transpose (jacobian leg)))
                     torques))))
             (if (still-in-contact leg resultant-force body)
               (add-forces-and-torques-to-body
                 body r resultant-force))))))

(defmethod add-forces-and-torques-to-body ((body aquarobot-body) r f)
  (let ((torques (cross-product r f)))
     (setf (forces-and-torques body)
         (vector-add (forces-and-torques body)
                 (append f torques)))))
```

```
;update joint angles and foot position. detect foot hitting ground.
(defmethod new-contact ((leg aqua-leg))
  (move-incremental leg '(0 0 0))
  (if (> (third (current-foot-position leg)) 0)
     (setf (third (current-foot-position leg)) 0
         (foot-contact leg) t)
     nil))

;detect loss of contact. (positive/down z component in world coord)
;side effect of reseting leg to default state when nil is returned.
(defmethod still-in-contact ((leg aqua-leg) force/body-xyz
                    (body aquarobot-body))
  (let ((force/world-xyz (vector-subtract (body-to-world body force/body-xyz)
                                (location body))))
     (if (> (third force/world-xyz) 0)
        (and (set-default-angles leg) (setf (foot-contact leg) nil))
        t)))
```

APPENDIX D

OPERATING INSTRUCTIONS

Call "droptest" with zero to four arguments.

First arg	Spring Constant (2..15, default 5)
Second arg	Damper Constant(0.5..15, default 5)
Third arg	Drop Height (0..100, default 0) cm
Fourth arg	Update Time Increment (10..50, default 50) ms

SOURCE CODE FILES

// file "droptest.c"

```
/*
/*  droptest.c
/*
/*   performer Aquarobot model with "spring" joints.
/*
/* */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <gl/device.h>

/* performer */
#include "pf.h"

/* performer aqua-robot object constructor */
#include "pf_aqua.h"

/* physical aqua-robot object constructor and controls */
#include "aqua.H"

static void OpenPipeline (pfPipe *p);
```

```
void
main (int argc, char *argv[])
{
   pfPipe     *p;
   pfChannel  *chan;
   pfScene    *scene;
   pfDCS      *robot_position;
   pfGroup    *aqua_robot;    /* graphics object (performer) */
   pfDCS      *JointDCS[6][4];
   aquarobot  robot;          /* physical object */

   // defaults for args
   float spring = SPRING_K;
   float damp   = SPRING_D;
   float height = 0.0f;
   float step   = 0.05f; // default ~ real time

   // process args
   if (argc > 1) { // spring constant
    // first arg: -15,000,000 <= spring <= -2,000,000
    spring = fabs((float)(atoi(argv[1])));
    if (spring < 2.0f)
      spring = 2.0f;
    else if (spring > 15.0f)
      spring = 15.0f;
    spring *= -1000000;
   }
   if (argc > 2) { // spring damping constant
    // second arg: -1,500,000 <= damp <= - 50,000
    damp = fabs((float)(atoi(argv[2])));
    if (damp < 0.5f)
      damp = 0.5f;
    else if (damp > 15.0f)
      damp = 15.0f;
    damp *= -100000;
   }
   if (argc > 3) {
    // third arg: 0 <= drop height <= 100
    height = fabs((float)(atoi(argv[3])));
    if (height > 100.0f)
      height = 100.0f;
   }
   if (argc > 4) {
    // fourth arg: 10ms <= integration time step <= 50ms
    step = fabs((float)(atoi(argv[4])))/1000.0f;
    if (step < 0.01f)
      step = 0.01f;
    else if (step > 0.05f)
      step = 0.05f;
   }
```

```
/* 1. initialize Performer */
pfInit();

/* 2. configure MP mode and start parallel processes */
pfConfig();

/* 3. load scene database */
scene = pfNewScene();

robot_position = pfNewDCS();
pfAddChild(scene, robot_position);
aqua_robot = MakeAquaRobot(JointDCS);
pfAddChild(robot_position, aqua_robot);

/* 5. configure and open full-screen pipeline */
p = pfGetPipe(0);
pfInitPipe(p, OpenPipeline); /* pfInitPipe(p, NULL); */

/* set frames per second ( if step = .05 sec, then ~ real time) */
pfFrameRate(20.0f);

/* 6. get and configure channel */
chan = pfNewChan(p);
pfChanScene(chan, scene);
pfChanNearFar(chan, 0.1f, 1000.0f);
pfChanFOV(chan, 45.0f, -1.0f);

/* zero clock (not really needed) */
pfInitClock();

/* initialize robot */
robot.initialize(spring, damp, height);
update_jointDCS(robot, JointDCS);

/* set up view position */
pfCoord  view;
pfSetVec3(view.hpr, 0.0f, 30.0f, 180.0f);
pfSetVec3(view.xyz, 0.0f, -500.0f, -350.0f);
pfChanView(chan, view.xyz, view.hpr);
```

```
    /* 7. create rendering loop */
    /* simulate for 30 seconds */
    int t = 0;
    while (t < 600) // ~ 20 frames per second
    {
       /* Transfer robot data to graphics object. */
       pfDCSMatrix(robot_position, robot.body.H_matrix); /* body   */
       update_jointDCS(robot, JointDCS);              /* joints */

      /* Go to sleep till next frame time */
      pfSync(); t++;

      /* initiate cull/draw for this frame */
      pfFrame();
      pfDrawChanStats(chan);

       /* Move robot to new position. */
       robot.update_aquarobot(step);
    }

    /* 8. terminate parallel processes and exit */
    pfExit();
    exit(0);
}
```

```
/*
 *  OpenPipeline() -- create a pipeline: setup the window system,
 *  the IRIS GL, and IRIS Performer. This procedure is executed in
 *  the draw process (when there is a separate draw process).
 */

static void
OpenPipeline (pfPipe *p)
{
   pfLight *Sun;

   /* negotiate with window-manager */
   foreground();
   prefposition(0,600,0,600);
   winopen("Aqua  Drop");

   /* negotiate with GL */
   pfInitGfx(p);

   /* create a light source */
   Sun = pfNewLight(pfGetSharedArena());
   pfLightPos(Sun, 0.0f, 0.0f, 1.0f, 0.0f);

   /* create a default lighting model */
   pfApplyLModel(pfNewLModel(pfGetSharedArena()));
   pfLightOn(Sun);
}
```

// file "pf_aqua.h"

```
/*
 * pf_aqua.h
 *
 * call "MakeAquaRobot" to make AquaRobot performer object.
 *
 * JointDCS[i][j] points to pfDCS for leg i, joint j,
 * where j = 0 is the shoulder joint, and j = 3 attaches the foot.
 *
 */

#include "pf.h"

pfGroup*
MakeAquaRobot(pfDCS *JointDCS[6][4]);
```

// file "pf_aqua.c"

```
/*
 * pf_aqua.c
 *
 * call "MakeAquaRobot" to make AquaRobot performer object.
 *
 */

#include "pf_aqua.h"
#include "aqua_link.H"

/* polygon data for aquarobot */
#include "polybody.h"
#include "polyshoulder.h"
#include "polyupperleg.h"
#include "polylowerleg.h"
#include "polyfoot.h"

/* geostate for multiple parts */
static pfGeoState *robotyellow_gstate;
```

```
pfGeoSet*
MakeBodyGSet(void)
{
   pfGeoSet  *gset;
   void      *arena;
   pfMaterial *mtl;

   arena = pfGetSharedArena();
   gset = pfNewGSet(arena);

   /* set the coordinate and normal arrays */
   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, bodycoords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, bodynorms, NULL);

   pfGSetPrimType(gset, PFGS_QUADS);
   pfGSetNumPrims(gset, 94);

   /* set up geostate for "robotyellow" material */
   robotyellow_gstate = pfNewGState(arena);
   mtl = pfNewMtl(arena);
   pfMtlColor(mtl, PFMTL_AMBIENT, 0.2f, 0.2f, 0.0f);
   pfMtlColor(mtl, PFMTL_DIFFUSE, 1.0f, 1.0f, 0.0f);
   pfMtlColor(mtl, PFMTL_EMISSION, 0.0f, 0.0f, 0.0f);
   pfMtlColor(mtl, PFMTL_SPECULAR, 0.0f, 0.0f, 0.0f);
   pfMtlAlpha(mtl, 1.0);
   pfGStateAttr(robotyellow_gstate, PFSTATE_FRONTMTL, mtl);
   pfGSetGState(gset, robotyellow_gstate);

   return gset;
}

pfGeoSet*
MakeLink1GSet(void)
{
   pfGeoSet *gset;
   void    *arena;

   arena = pfGetSharedArena();
   gset  = pfNewGSet(arena);

   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, link1coords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, link1norms, NULL);

   pfGSetPrimType(gset, PFGS_QUADS);
   pfGSetNumPrims(gset, 42);

   pfGSetGState(gset, robotyellow_gstate);

   return gset;
}
```

```
pfGeoSet*
MakeLink2GSet(void)
{
   pfGeoSet *gset;
   void    *arena;

   arena = pfGetSharedArena();
   gset  = pfNewGSet(arena);

   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, link2coords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, link2norms, NULL);

   pfGSetPrimType(gset, PFGS_QUADS);
   pfGSetNumPrims(gset, 91);

   pfGSetGState(gset, robotyellow_gstate);

   return gset;
}

pfGeoSet*
MakeLink3GSet(void)
{
   pfGeoSet *gset;
   void    *arena;

   arena = pfGetSharedArena();
   gset = pfNewGSet(arena);

   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, link3coords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, link3norms, NULL);

   pfGSetPrimType(gset, PFGS_QUADS);
   pfGSetNumPrims(gset, 103);

   pfGSetGState(gset, robotyellow_gstate);

   return gset;
}
```

```
pfGeoSet*
MakeFootGSet(void)
{
   pfGeoSet *gset;
   void    *arena;

   arena = pfGetSharedArena();
   gset  = pfNewGSet(arena);

   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, footcoords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, footnorms, NULL);

   pfGSetPrimType(gset, PFGS_TRISTRIPS);
   pfGSetNumPrims(gset, 49);
   pfGSetPrimLengths(gset, footstriplengths);

   pfGSetGState(gset, robotyellow_gstate);

   return gset;
}

pfGeoSet*
MakeShaftGSet(void)
{
   pfGeoSet   *gset;
   void       *arena;
   pfGeoState *robotgray_gstate;
   pfMaterial *mtl;
   arena = pfGetSharedArena();
   gset = pfNewGSet(arena);

   pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX, shaftcoords, NULL);
   pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM, shaftnorms, NULL);

   pfGSetPrimType(gset, PFGS_QUADS);
   pfGSetNumPrims(gset, 20);

   /* set up material */
   robotgray_gstate = pfNewGState(arena);
   mtl = pfNewMtl(arena);
   pfMtlColor(mtl, PFMTL_AMBIENT, 0.1, 0.1, 0.1);
   pfMtlColor(mtl, PFMTL_DIFFUSE, 0.2, 0.2, 0.2);
   pfMtlColor(mtl, PFMTL_EMISSION, 0.0, 0.0, 0.0);
   pfMtlColor(mtl, PFMTL_SPECULAR, 0.0, 0.0, 0.0);
   pfMtlAlpha(mtl, 1.0);
   pfGStateAttr(robotgray_gstate, PFSTATE_FRONTMTL, mtl);
   pfGSetGState(gset, robotgray_gstate);

   return gset;
}
```

```
pfGroup*
MakeAquaRobot(pfDCS *JointDCS[6][4])
{
  pfSCS     *LegAttachSCS[6],
            *Link1SCS[6], *Link2SCS[6], *Link3SCS[6],
            *FootSCS[6];
  pfMatrix   rot_mat, trans_mat;
  pfGroup    *AquaRobotGroup, *BodyGroup[6], *LegGroup[6];
  pfGeode    *geode_body,
            *geode_link1, *geode_link2, *geode_link3,
            *geode_shaft, *geode_foot;
  int        i; /* loop counter */

  /* make geodes */
  geode_body  = pfNewGeode();
  pfAddGSet(geode_body, MakeBodyGSet());

  geode_link1 = pfNewGeode();
  pfAddGSet(geode_link1, MakeLink1GSet());

  geode_link2 = pfNewGeode();
  pfAddGSet(geode_link2, MakeLink2GSet());

  geode_link3 = pfNewGeode();
  pfAddGSet(geode_link3, MakeLink3GSet());

  geode_foot = pfNewGeode();
  pfAddGSet(geode_foot, MakeFootGSet());

  geode_shaft = pfNewGeode();
  pfAddGSet(geode_shaft, MakeShaftGSet());

  /* Make Parent Group */
  AquaRobotGroup = pfNewGroup();

  /* Add Structure (6 segments) */
  for(i = 0; i < 6; i++)
  {
    /* rotate to segment */
    pfMakeRotMat(rot_mat, i*60.0, 0.0, 0.0, 1.0);
    LegAttachSCS[i] = pfNewSCS(rot_mat);
    pfAddChild(AquaRobotGroup, LegAttachSCS[i]);

    /* add body slice */
    BodyGroup[i] = pfNewGroup();
    pfAddChild(LegAttachSCS[i], BodyGroup[i]);
    pfAddChild(BodyGroup[i], geode_body);
```

```
    /* add leg */

    /* link 1 */
    pfMakeTransMat(trans_mat, LINK0LENGTH, 0.0, 0.0);
    Link1SCS[i] = pfNewSCS(trans_mat);
    pfAddChild(BodyGroup[i], Link1SCS[i]);
    pfAddChild(Link1SCS[i], geode_shaft);

    JointDCS[i][0] = pfNewDCS();
    pfAddChild(Link1SCS[i], JointDCS[i][0]);
    pfAddChild(JointDCS[i][0], geode_link1);

    /* link 2 */
    pfMakeRotMat(rot_mat, -90.0, 1.0, 0.0, 0.0);
    pfMakeTransMat(trans_mat, LINK1LENGTH, 0.0, 0.0);
    pfPostMultMat(rot_mat, trans_mat);
    Link2SCS[i] = pfNewSCS(rot_mat);
    pfAddChild(JointDCS[i][0], Link2SCS[i]);
    pfAddChild(Link2SCS[i], geode_shaft);

    JointDCS[i][1] = pfNewDCS();
    pfAddChild(Link2SCS[i], JointDCS[i][1]);
    pfAddChild(JointDCS[i][1], geode_link2);

    /* link 3 */
    pfMakeTransMat(trans_mat, LINK2LENGTH, 0.0, 0.0);
    Link3SCS[i] = pfNewSCS(trans_mat);
    pfAddChild(JointDCS[i][1], Link3SCS[i]);
    pfAddChild(Link3SCS[i], geode_shaft);

    JointDCS[i][2] = pfNewDCS();
    pfAddChild(Link3SCS[i], JointDCS[i][2]);
    pfAddChild(JointDCS[i][2], geode_link3);

    /* foot */
    pfMakeTransMat(trans_mat, LINK3LENGTH, 0.0, 0.0);
    FootSCS[i] = pfNewSCS(trans_mat);
    pfAddChild(JointDCS[i][2], FootSCS[i]);

    JointDCS[i][3] = pfNewDCS();
    pfAddChild(FootSCS[i], JointDCS[i][3]);
    pfAddChild(JointDCS[i][3], geode_foot);
  }
  return AquaRobotGroup;
}
```

```
// file "aqua.h"


#ifndef _AQUA_H
#define _AQUA_H

#include <Performer/pf.h>
#include "misc.H"
#include "rigid_body.H"
#include "aqua_leg.H"

typedef rigid_body aquarobot_body;

/* mass in Kg. */
#define AQUA_BODY_MASS 500.0f

/* (Ix Iy Iz)-Kg-cm.  in principal axis coordinates. */
/* assumes solid cylindrical body of constant density. */
#define AQUA_BODY_HEIGHT 50.0f
#define AQUA_BODY_RADIUS 30.0f
static pfVec3 aqua_body_inertia = {
    // Ix
      1.0f/4.0f  * AQUA_BODY_MASS * AQUA_BODY_RADIUS * AQUA_BODY_RADIUS
    + 1.0f/12.0f * AQUA_BODY_MASS * AQUA_BODY_HEIGHT * AQUA_BODY_HEIGHT,
    // Iy
      1.0f/4.0f  * AQUA_BODY_MASS * AQUA_BODY_RADIUS * AQUA_BODY_RADIUS
    + 1.0f/12.0f * AQUA_BODY_MASS * AQUA_BODY_HEIGHT * AQUA_BODY_HEIGHT,
    // Iz
      1.0f/2.0f  * AQUA_BODY_MASS * AQUA_BODY_RADIUS * AQUA_BODY_RADIUS};

/* initial position and orientation in world coordinates. */
#define AZIMUTH_INIT    deg_to_rad(0.0f)
#define ELEVATION_INIT  deg_to_rad(0.0f)
#define ROLL_INIT       deg_to_rad(0.0f)
#define X_INIT 0.0f
#define Y_INIT 0.0f
#define Z_INIT sinf(default_angles[1])*LINK2LENGTH - LINK3LENGTH

/* leg attachment angles. */
#define LEG1ANGLE   deg_to_rad(0.0f)
#define LEG2ANGLE   deg_to_rad(60.0f)
#define LEG3ANGLE   deg_to_rad(120.0f)
#define LEG4ANGLE   deg_to_rad(180.0f)
#define LEG5ANGLE   deg_to_rad(240.0f)
#define LEG6ANGLE   deg_to_rad(300.0f)
```

```
class aquarobot
{
public:
 aquarobot_body body;
//private:
 aqua_leg      leg1, leg2, leg3, leg4, leg5, leg6;

private:
 void
 aquarobot::init_joint_access();

 void
 update_forces_and_torques();

 void
 update_legs();

public:
 // External access to joint angles for passing to performer model
 // This could be private if "update_jointDCS" were a friend;
 // however, the class should not depend on needs of user.
 float*     joint_matrix[6][4];

public:
 aquarobot():body(AQUA_BODY_MASS, aqua_body_inertia),
         leg1(LEG1ANGLE),
         leg2(LEG2ANGLE),
         leg3(LEG3ANGLE),
         leg4(LEG4ANGLE),
         leg5(LEG5ANGLE),
         leg6(LEG6ANGLE) {init_joint_access();}

 void
 initialize(float k = SPRING_K, float d = SPRING_D, float height = 0.0f);

 void
 update_aquarobot(float dt = 0.0f);

 // coordinate transformation routines
 void
 world_to_aqua(pfVec3 destination, pfVec3 source)
         {body.world_to_body(destination, source);}

 void
 aqua_to_world(pfVec3 destination, pfVec3 source)
         {body.body_to_world(destination, source);}
};
```

```
void
update_jointDCS(aquarobot robot, pfDCS *JointDCS[6][4]);

#endif
```

// File "aqua.c"

```
#include "aqua.H"
#include <math.h>

/* user routines */

void
aquarobot::initialize(float k, float d, float height)
{
 body.move(AZIMUTH_INIT, ELEVATION_INIT, ROLL_INIT,
        X_INIT, Y_INIT, -fabs(height)+Z_INIT);
 pfSetVec3(body.vel_trans, 0.0f, 0.0f, 0.0f);
 pfCopyVec3(body.vel_rot, body.vel_trans);
 pfCopyVec3(body.accel_trans, body.vel_trans);
 pfCopyVec3(body.accel_rot, body.vel_trans);
 pfCopyVec3(body.forces, body.vel_trans);
 pfCopyVec3(body.torques, body.vel_trans);
 body.start_timer();
 leg1.init_leg(&body, k, d);
 leg2.init_leg(&body, k, d);
 leg3.init_leg(&body, k, d);
 leg4.init_leg(&body, k, d);
 leg5.init_leg(&body, k, d);
 leg6.init_leg(&body, k, d);
 update_forces_and_torques();
}

void aquarobot::update_aquarobot(float dt)
{
 float dt_ = dt;
 if (dt <= 0.0)
  dt_ = body.get_delta_t(); // default

 body.update_acceleration();
 body.update_velocity(dt_);
 body.update_H_matrix(dt_);
 body.update_position(dt_);
//  body.update_velocity(dt_);
 update_legs();
 /* This is done last as it also updates leg positions */
 /* leg positions depend on "new" body position!      */
 update_forces_and_torques();
}
```

```
/* Internal Routines */

void
aquarobot::init_joint_access()
{
 // for use, see fn:"update_jointDCS" below
 joint_matrix[0][0] = &leg1.link1.inboard_joint_angle;
 joint_matrix[0][1] = &leg1.link2.inboard_joint_angle;
 joint_matrix[0][2] = &leg1.link3.inboard_joint_angle;
 joint_matrix[0][3] = &leg1.link4.H_matrix[0][0];

 joint_matrix[1][0] = &leg2.link1.inboard_joint_angle;
 joint_matrix[1][1] = &leg2.link2.inboard_joint_angle;
 joint_matrix[1][2] = &leg2.link3.inboard_joint_angle;
 joint_matrix[1][3] = &leg2.link4.H_matrix[0][0];

 joint_matrix[2][0] = &leg3.link1.inboard_joint_angle;
 joint_matrix[2][1] = &leg3.link2.inboard_joint_angle;
 joint_matrix[2][2] = &leg3.link3.inboard_joint_angle;
 joint_matrix[2][3] = &leg3.link4.H_matrix[0][0];

 joint_matrix[3][0] = &leg4.link1.inboard_joint_angle;
 joint_matrix[3][1] = &leg4.link2.inboard_joint_angle;
 joint_matrix[3][2] = &leg4.link3.inboard_joint_angle;
 joint_matrix[3][3] = &leg4.link4.H_matrix[0][0];

 joint_matrix[4][0] = &leg5.link1.inboard_joint_angle;
 joint_matrix[4][1] = &leg5.link2.inboard_joint_angle;
 joint_matrix[4][2] = &leg5.link3.inboard_joint_angle;
 joint_matrix[4][3] = &leg5.link4.H_matrix[0][0];

 joint_matrix[5][0] = &leg6.link1.inboard_joint_angle;
 joint_matrix[5][1] = &leg6.link2.inboard_joint_angle;
 joint_matrix[5][2] = &leg6.link3.inboard_joint_angle;
 joint_matrix[5][3] = &leg6.link4.H_matrix[0][0];
}

void aquarobot::update_forces_and_torques()
{
 pfSetVec3(body.forces, 0.0f, 0.0f, 0.0f);
 pfSetVec3(body.torques, 0.0f, 0.0f, 0.0f);
 leg1.add_leg_forces_and_torques();
 leg2.add_leg_forces_and_torques();
 leg3.add_leg_forces_and_torques();
 leg4.add_leg_forces_and_torques();
 leg5.add_leg_forces_and_torques();
 leg6.add_leg_forces_and_torques();
}
```

```
void aquarobot::update_legs()
{
 leg1.update_leg();
 leg2.update_leg();
 leg3.update_leg();
 leg4.update_leg();
 leg5.update_leg();
 leg6.update_leg();
}


/* joint angle transfer routine */

void
update_jointDCS(aquarobot robot, pfDCS *JointDCS[6][4])
{
static pfMatrix m4 = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,1}};
 for(int i=0;i<6;i++) {
   // rotate first three joints
   for(int j=0;j<3;j++) {
     pfDCSRot(JointDCS[i][j],
           rad_to_deg(*robot.joint_matrix[i][j]), 0.0f, 0.0f);
   }
   // equiv to rot(0x,-90y,0z) * inverse(leg[i].link4.H_Matrix)
   // accomplishes DCS such that link(x-axis) || world(z-axis)
   m4[0][0] =  (robot.joint_matrix[i][3])[2];
   m4[0][1] =  (robot.joint_matrix[i][3])[6];
   m4[0][2] =  (robot.joint_matrix[i][3])[10];
   m4[1][0] =  (robot.joint_matrix[i][3])[1];
   m4[1][1] =  (robot.joint_matrix[i][3])[5];
   m4[1][2] =  (robot.joint_matrix[i][3])[9];
   m4[2][0] = -(robot.joint_matrix[i][3])[0];
   m4[2][1] = -(robot.joint_matrix[i][3])[4];
   m4[2][2] = -(robot.joint_matrix[i][3])[8];
   pfDCSMatrix(JointDCS[i][3], m4);
 }
}
```

```
// file "aqua_leg.h"

#ifndef _AQUA_LEG_H
#define _AQUA_LEG_H

#include <Performer/pf.h>
#include "rigid_body.H"
#include "aqua_link.H"
#include "misc.H"

/* initial (default) joint angles. */
static pfVec3 default_angles = {
   /*   0 deg */   0.0f,
   /*  45 deg */  25.0f * PI_F / 180.0f,
   /* -135 deg */ -115.0f * PI_F / 180.0f};

/* joint spring constant. (default : 5,000,000 Kg-cm2/sec2 per radian) */
#define SPRING_K -5000000.0f

/* joint spring damping constant. (default : 500,000 Kg-cm2/sec2 per radian/sec) */
#define SPRING_D  -500000.0f

/* AQUA LEG CLASS */

class aqua_leg
{
public:
 float     leg_attachment_angle;
 aqualink0  link0;
 aqualink1  link1;
 aqualink2  link2;
 aqualink3  link3;
 aqualink4  link4;
 boolean    motion_complete_flag;
 pfVec3     previous_foot_position;
 pfVec3     current_foot_position;
 boolean    foot_contact;
 float      spr_k;
 float      spr_d;

public:
 aqua_leg(float angle = 0.0f): leg_attachment_angle(angle)
       {
        motion_complete_flag = TRUE;
        foot_contact = FALSE;
       }

 void
 init_leg(rigid_body *body, float k = SPRING_K, float d = SPRING_D);
```

```
 void
 set_default_angles();

 void
 update_leg();

 void
 update_foot_pos();

 void
 set_angles(float joint1, float joint2, float joint3);

 void
 set_angles(pfVec3 angles) {set_angles(angles[0], angles[1], angles[2]);}

 void
 jacobian(pfMatrix J);

 void
 inverse_jacobian(pfMatrix J_inv);

 void
 joint_rates(pfVec3 rates);

 void
 FootPosFmJoint1(pfVec3 foot_pos_j1, pfVec3 foot_pos_world);

 void
 aqua_inv_kin(pfVec3 joint_angles, pfVec3 world_foot_pos);

 void
 add_leg_forces_and_torques();

 int
 new_contact();

 int
 still_in_contact(pfVec3 leg_force_body);
};

#endif
```

// file "aqua_leg.c"

```
#include <math.h>
#include "aqua_leg.H"

/* AQUA-ROBOT INVERSE KINEMATICS ROUTINES */

static float L2sqr = LINK2LENGTH * LINK2LENGTH;
static float L3sqr = LINK3LENGTH * LINK3LENGTH;

/* routines that return the joint angles for a leg, given the foot position */

float
theta1(pfVec3 foot_pos)
{
 if (foot_pos[0] < 0.0f)
   return (atan2f(-foot_pos[1], -foot_pos[0]));
 else
   return (atan2f( foot_pos[1],  foot_pos[0]));
}

float
theta2(pfVec3 foot_pos, float hyp, float hyp_sqr)
{
 float temp = asinf(foot_pos[2]/hyp);
 if (foot_pos[0] < 0.0f) temp = PI_F - temp;
 return (acosf((L2sqr + hyp_sqr - L3sqr) / (2 * LINK2LENGTH * hyp)) - temp );
}

float
theta3(float hyp_sqr)
{
 return (acosf((L2sqr + L3sqr - hyp_sqr) / (2 * LINK2LENGTH * LINK3LENGTH)) - PI_F);
}

/* supports theta2 and theta3 which require foot position with respect */
/* to joint2 position. joint2 position depends on theta1.          */
void
FootPosFmJoint2(pfVec3 destination, pfVec3 source)
{
 destination[0] = sqrtf(source[0]*source[0] + source[1]*source[1])
                - LINK1LENGTH;
 destination[1] = 0.0f;
 destination[2] = source[2];
}
```

```
/* AQUA LEG CLASS */

void
aqua_leg::init_leg(rigid_body *body, float k, float d)
{
 spr_k = k; spr_d = d;
 foot_contact = 0;
 link0.inboard_link = body;
 link1.inboard_link = &link0;
 link2.inboard_link = &link1;
 link3.inboard_link = &link2;
 link4.inboard_link = &link3;
 set_default_angles();
}

void
aqua_leg::set_default_angles()
{
 set_angles(default_angles);
 update_foot_pos();
 pfCopyVec3(previous_foot_position, current_foot_position);
}

void
aqua_leg::update_leg()
{
 pfCopyVec3(previous_foot_position, current_foot_position);
 set_angles(link1.inboard_joint_angle,
        link2.inboard_joint_angle,
        link3.inboard_joint_angle);
 update_foot_pos();
}

void
aqua_leg::update_foot_pos()
{
 if (!foot_contact) {
   current_foot_position[0] = link4.H_matrix[3][0];
   current_foot_position[1] = link4.H_matrix[3][1];
   current_foot_position[2] = link4.H_matrix[3][2];
  }
}
```

```
void
aqua_leg::set_angles(float joint1, float joint2, float joint3)
{
 link0.rotate_link(leg_attachment_angle);
 link1.rotate_link(joint1);
 link2.rotate_link(joint2);
 link3.rotate_link(joint3);
// this works for each leg in 2D, but world 3D solution requires(0.0)
// link4.rotate_link(-deg_to_rad(90.0f) - joint2 - joint3);
 link4.rotate_link(0.0f);
}

void
aqua_leg::jacobian(pfMatrix J)
{
 pfVec3 row;
 float  angle, S01, C01, S2, C2, S23, C23;
 #define L1 LINK1LENGTH
 #define L2 LINK2LENGTH
 #define L3 LINK3LENGTH

 angle = leg_attachment_angle + link1.inboard_joint_angle;
 S01 = sinf(angle);
 C01 = cosf(angle);
 angle = link2.inboard_joint_angle;
 S2 = sinf(angle);
 C2 = cosf(angle);
 angle += link3.inboard_joint_angle;
 S23 = sinf(angle);
 C23 = cosf(angle);

 pfMakeIdentMat(J);
 pfSetVec3(row, -S01 * (L1 + L2*C2 + L3*C23),
           -C01 * (L2*S2 + L3*S23),
           -C01 * L3 * S23);
 pfSetMatColVec3(J, 0, row);
 pfSetVec3(row,  C01 * (L1 + L2*C2 + L3*C23),
           -S01 * (L2*S2 + L3*S23),
           -S01 * L3 * S23);
 pfSetMatColVec3(J, 1, row);
 pfSetVec3(row, 0.0f,
           -L2*C2 - L3*C23,
           -L3 * C23);
 pfSetMatColVec3(J, 2, row);
}
```

```
void
aqua_leg::inverse_jacobian(pfMatrix J_inv)
{
 pfMatrix J;
 jacobian(J);
 pfInvertMat(J_inv, J);
}

void
aqua_leg::joint_rates(pfVec3 rates)
{
 pfMatrix J_inv;
 pfVec3   trans_rates;
 pfVec3   omega;
 pfVec3   foot_r;
 pfVec3   rot_rates;
 pfVec3   foot_rates;

 inverse_jacobian(J_inv);

 pfScaleVec3(trans_rates, -1.0f, ((rigid_body *)link0.inboard_link)->vel_trans);

 pfCopyVec3(omega, ((rigid_body *)link0.inboard_link)->vel_rot);
 ((rigid_body *)link0.inboard_link)->world_to_body(foot_r, current_foot_position);
 pfCrossVec3(rot_rates, omega, foot_r);

 pfSubVec3( foot_rates, trans_rates, rot_rates);
 post_mult(rates, J_inv, foot_rates);
}

void
aqua_leg::FootPosFmJoint1(pfVec3 foot_pos_j1, pfVec3 foot_pos_world)
{
 link0.world_to_body(foot_pos_j1, foot_pos_world);
 foot_pos_j1[0] -= LINK0LENGTH;
}

void
aqua_leg::aqua_inv_kin(pfVec3 joint_angles, pfVec3 world_foot_pos)
{
 pfVec3 foot_joint1, foot_joint2;
 float  hyp, hyp_sqr;
 FootPosFmJoint1(foot_joint1, world_foot_pos);
 FootPosFmJoint2(foot_joint2, foot_joint1);
 hyp = pfLengthVec3(foot_joint2);
 hyp_sqr = hyp * hyp;
 pfSetVec3(joint_angles, theta1(foot_joint1),
                         theta2(foot_joint2, hyp, hyp_sqr),
                         theta3(hyp_sqr));
}
```

```
void
aqua_leg::add_leg_forces_and_torques()
{
 if (foot_contact || new_contact())
   {
    pfVec3   angles, joint_torques, damp, forces, foot_pos;
    pfMatrix work_matrix1, work_matrix2;

    aqua_inv_kin(angles, current_foot_position);
    set_angles(angles);

    /* get spring force of joints */
    pfSubVec3(joint_torques, angles, default_angles);
    pfScaleVec3(joint_torques, spr_k, joint_torques);

    /* add damping */
    joint_rates(damp);
    pfScaleVec3(damp, spr_d, damp);
    pfAddVec3(joint_torques, joint_torques, damp);

    jacobian(work_matrix1);
    pfTransposeMat(work_matrix2, work_matrix1);
    pfInvertMat(work_matrix1, work_matrix2);
    post_mult(forces, work_matrix1, joint_torques);
    pfScaleVec3(forces, -1.0, forces);

    if (still_in_contact(forces))
     {
      ((rigid_body *)link0.inboard_link)->world_to_body(foot_pos, current_foot_position);
      ((rigid_body *)link0.inboard_link)->add_force_and_torques(foot_pos, forces);
     }
   }
}

int
aqua_leg::new_contact()
{
 if (current_foot_position[2] > 0.0f)
   {
    current_foot_position[2] = 0.0f;
    return (foot_contact = TRUE);
   }
 else
   return FALSE;
}
```

```
int
aqua_leg::still_in_contact(pfVec3 leg_force_body)
{
 pfVec3 leg_force_world;
 ((rigid_body *)link0.inboard_link)->body_to_world(leg_force_world, leg_force_body);
 pfSubVec3(leg_force_world, leg_force_world, ((rigid_body *)link0.inboard_link)->location);
 if (leg_force_world[2] > 0.0f)
   {
    set_default_angles();
    return (foot_contact = FALSE);
   }
 else
   return TRUE;
}
```

```
// file "aqua_link.h"


#ifndef _AQUA_LINK_H_
#define _AQUA_LINK_H_

#include <Performer/pf.h>
#include "rigid_body.H"

/* BASE CLASSES */

class link:public rigid_body
{
public:
  int      motion_limit_flag;
  float    length_i_1;
  float    twist_i_1;
  float    inboard_joint_angle;
  float    inboard_joint_displacement;
  void*    inboard_link;
  pfMatrix  T_matrix;

public:
  link(float mass = 1.0f, pfVec3 moments = NULL):rigid_body(mass, moments){}
  void update_T_matrix();
  void rotate(float angle);
};

class rotary_link:public link
{
public:
  float min_joint_angle;
  float max_joint_angle;

public:
  rotary_link(float length          = 0.0f,
              float min_angle       = 0.0f,
              float max_angle       = 0.0f,
              float twist           = 0.0f,
              float joint_angle     = 0.0f,
              float joint_displacement = 0.0f,
              void* inboard_link    = 0);
  void rotate_link(float angle);
};
```

```
/* MODIFIED DANEVIT-HARTENBERG LINK COORDINATE TRANSFORMATION MATRIX */

void
mdh_matrix(pfMatrix mdh,
        float cosrotate,    float sinrotate,
        float costwist_i_1, float sintwist_i_1,
        float length_i_1,   float translate);


/* AQUA-LINK CLASSES */

/* link lengths */
#define LINK0LENGTH 37.5f
#define LINK1LENGTH 20.0f
#define LINK2LENGTH 52.0f
#define LINK3LENGTH 102.0f
#define LINK4LENGTH 3.0f

/* joint limits */
#define JOINT1MIN deg_to_rad( -60.0f)
#define JOINT1MAX deg_to_rad(  60.0f)
#define JOINT2MIN deg_to_rad(-360.0f)
#define JOINT2MAX deg_to_rad( 360.0f)
#define JOINT3MIN deg_to_rad(-360.0f)
#define JOINT3MAX deg_to_rad( 360.0f)
#define JOINT4MIN deg_to_rad( -22.5f)
#define JOINT4MAX deg_to_rad(  22.5f)

class aqualink0:public rotary_link
{
public:
  aqualink0();
};

class aqualink1:public rotary_link
{
public:
  aqualink1();
};

class aqualink2:public rotary_link
{
public:
  aqualink2();
};
```

```
class aqualink3:public rotary_link
{
public:
  aqualink3();
};

class aqualink4:public rotary_link
{
public:
  aqualink4();
};

#endif
```

```
// file "aqua_link.c"


#include <math.h>
#include "aqua_link.H"
#include "misc.H"

/* BASE CLASSES */

void
link::update_T_matrix()
{
 float sa = sinf(inboard_joint_angle);
 float ca = cosf(inboard_joint_angle);
 float st = sinf(twist_i_1);
 float ct = cosf(twist_i_1);

 mdh_matrix(T_matrix, ca, sa, ct, st, length_i_1, inboard_joint_displacement);
}

void
link::rotate(float angle)
{
 inboard_joint_angle = angle;
 update_T_matrix();
 // multiplied in reverse order as they are stored as transposes.
 pfMultMat(H_matrix, T_matrix, ((rigid_body *)inboard_link)->H_matrix);
}

rotary_link::rotary_link(float length,
                         float min_angle,
                         float max_angle,
                         float twist,
                         float joint_angle,
                         float joint_displacement,
                         void* inboard_link_)
{
 length_i_1             = length;
 min_joint_angle        = min_angle;
 max_joint_angle        = max_angle;
 twist_i_1              = twist;
 inboard_joint_angle    = joint_angle;
 inboard_joint_displacement = joint_displacement;
 inboard_link           = inboard_link_;
 pfMakeIdentMat(T_matrix);
}
```

```
void
rotary_link::rotate_link(float angle)
{
/* joint stops disabled.
 if (angle < min_joint_angle)
  {
   rotate(min_joint_angle);
   motion_limit_flag = -1;
  }
 else if (angle > max_joint_angle)
  {
   rotate(max_joint_angle);
   motion_limit_flag = 1;
  }
 else
  {
*/
   rotate(angle);
   motion_limit_flag = 0;
/*
  }
*/
}

void
mdh_matrix (pfMatrix mdh,
        float cosrotate,   float sinrotate,
        float costwist_i_1, float sintwist_i_1,
        float length_i_1,   float translate)
{
 /* col 1 */
 mdh[0][0] = cosrotate;
 mdh[1][0] = - sinrotate;
 mdh[2][0] = 0.0f;
 mdh[3][0] = length_i_1;
 /* col 2 */
 mdh[0][1] = sinrotate * costwist_i_1;
 mdh[1][1] = cosrotate * costwist_i_1;
 mdh[2][1] = - sintwist_i_1;
 mdh[3][1] = - sintwist_i_1 * translate;
 /* col 3 */
 mdh[0][2] = sinrotate * sintwist_i_1;
 mdh[1][2] = cosrotate * sintwist_i_1;
 mdh[2][2] = costwist_i_1;
 mdh[3][2] = costwist_i_1 * translate;
 /* col 4 */
 mdh[0][3] = mdh[1][3] = mdh[2][3] = 0.0f;
 mdh[3][3] = 1.0f;
```

```
/* alternate method using pf functions
 pfVec3  col;
 pfMakeIdentMat(mdh);
 pfSetVec3(col, cosrotate, sinrotate * costwist_i_1,
          sinrotate * sintwist_i_1);
 pfSetMatRowVec3(mdh, 0, col);
 pfSetVec3(col, -sinrotate,  cosrotate * costwist_i_1,
          cosrotate * sintwist_i_1);
 pfSetMatRowVec3(mdh, 1, col);
 pfSetVec3(col, 0.0, -sintwist_i_1, costwist_i_1);
 pfSetMatRowVec3(mdh, 2, col);
 pfSetVec3(col, length_i_1, - sintwist_i_1 * translate,
          costwist_i_1 * translate);
 pfSetMatRowVec3(mdh, 3, col);
*/
}

/* AQUA-LINK CLASSES */

aqualink0::aqualink0()
{
 max_joint_angle = deg_to_rad(360.0f);
}

aqualink1::aqualink1()
{
 length_i_1     = LINK0LENGTH;
 min_joint_angle = JOINT1MIN;
 max_joint_angle = JOINT1MAX;
}

aqualink2::aqualink2()
{
 length_i_1     = LINK1LENGTH;
 twist_i_1      = deg_to_rad(-90.0f);
 min_joint_angle = JOINT2MIN;
 max_joint_angle = JOINT2MAX;
}

aqualink3::aqualink3()
{
 length_i_1     = LINK2LENGTH;
 min_joint_angle = JOINT3MIN;
 max_joint_angle = JOINT3MAX;
}
```

```
aqualink4::aqualink4()
{
 length_i_1     = LINK3LENGTH;
 min_joint_angle = JOINT4MIN;
 max_joint_angle = JOINT4MAX;
}
```

```
// file "rigid_body.H"


#ifndef _RIGID_BODY_
#define _RIGID_BODY_

#include <Performer/pf.h>

#define GRAVITY 980.0

class rigid_body
{
public:
 pfVec3 location;  /* The vector (x y z) in world coordinates. */
 pfVec3 vel_trans;  /* The vector (u v w) in body coordinates. */
 pfVec3 vel_rot;   /* The vector (p q r) in body coordinates. */
 pfVec3 accel_trans;/* The vector (u-dot v-dot w-dot). */
 pfVec3 accel_rot;  /* The vector (p-dot q-dot r-dot). */
 pfVec3 forces;    /* The vector (Fx Fy Fz) in body coordinates. */
 pfVec3 torques;   /* The vector (L M N) in body coordinates. */
 pfVec3 moments_;  /* The vector (Ix Iy Iz) in principal axis coordinates. */
 float  mass_;     /* in Kg. */
 float  current_time;
 pfMatrix H_matrix;

public:
 rigid_body(float mass, pfVec3 moments = NULL);

 void
 move(float azimuth, float elevation, float roll,
    float x,      float y,      float z);

 float
 get_delta_t();

 void
 start_timer();

/*
 void
 update_rigid_body();
*/

 void
 update_acceleration();

 void
 update_velocity(float dt);
```

```
  void
  update_H_matrix(float dt);

  void
  update_position(float dt);

  void
  world_to_body(pfVec3 destination, pfVec3 source);

  void
  body_to_world(pfVec3 destination, pfVec3 source);

  void
  add_force_and_torques(pfVec3 r, pfVec3 f);

};

void
homogeneous_transform(pfMatrix homo,
                      float azimuth, float elevation, float roll,
                      float x,    float y,    float z);

void
post_mult(pfVec3 destination, pfMatrix m, pfVec3 source);

#endif
```

// file "rigid_body.C"

```
#include "rigid_body.H"
#include "misc.H"

rigid_body::rigid_body(float mass, pfVec3 moments)
{
 location  [0] = location  [1] = location  [2] =
 vel_trans [0] = vel_trans [1] = vel_trans [2] =
 vel_rot   [0] = vel_rot   [1] = vel_rot   [2] =
 accel_trans[0] = accel_trans[1] = accel_trans[2] =
 accel_rot [0] = accel_rot [1] = accel_rot [2] =
 forces    [0] = forces    [1] = forces    [2] =
 torques   [0] = torques   [1] = torques   [2] = 0.0f;
 if (moments == NULL)
  moments_[0] = moments_[1] = moments_[2] = 0.0f;
 else
  pfCopyVec3(moments_, moments);
 mass_ = mass;
 pfMakeIdentMat(H_matrix);
 current_time = 0.0f;
}

void
rigid_body::move(float azimuth, float elevation, float roll,
                 float x,       float y,         float z)
{
 homogeneous_transform(H_matrix, azimuth, elevation, roll, x, y, z);
 pfSetVec3(location, x, y, z);
}

float
rigid_body::get_delta_t()
{
 float dt = 0.05f;
 current_time += dt;
 return dt;
}

void
rigid_body::start_timer()
{
 current_time = 0.0f;
}
```

```
/*
void
rigid_body::update_rigid_body()
{
 float dt = get_delta_t;
 update_H_matrix(dt);
 update_position(dt);
 update_velocity(dt);
 update_acceleration();
}
*/

void
rigid_body::update_acceleration()
{
 accel_trans[0] = vel_trans[1] * vel_rot[2] - vel_trans[2] * vel_rot[1]
          + forces[0] / mass_       + GRAVITY * H_matrix[0][2];
 accel_trans[1] = vel_trans[2] * vel_rot[0] - vel_trans[0] * vel_rot[2]
          + forces[1] / mass_       + GRAVITY * H_matrix[1][2];
 accel_trans[2] = vel_trans[0] * vel_rot[1] - vel_trans[1] * vel_rot[0]
          + forces[2] / mass_       + GRAVITY * H_matrix[2][2];
 accel_rot[0]  =
     ((moments_[1] - moments_[2]) * vel_rot[1] * vel_rot[2] + torques[0])
                / moments_[0];
 accel_rot[1]  =
     ((moments_[2] - moments_[0]) * vel_rot[2] * vel_rot[0] + torques[1])
                / moments_[1];
 accel_rot[2]  =
     ((moments_[0] - moments_[1]) * vel_rot[0] * vel_rot[1] + torques[2])
                / moments_[2];
}

void
rigid_body::update_velocity(float dt)
{
 pfVec3 dv;
 pfScaleVec3(dv, dt, accel_trans);
 pfAddVec3(vel_trans, vel_trans, dv);
 pfScaleVec3(dv, dt, accel_rot);
 pfAddVec3(vel_rot, vel_rot, dv);
}
```

```
void
rigid_body::update_H_matrix(float dt)
{
 pfMatrix homo;
 homogeneous_transform(homo, dt * vel_rot[2],
                       dt * vel_rot[1],
                       dt * vel_rot[0],
                       dt * vel_trans[0],
                       dt * vel_trans[1],
                       dt * vel_trans[2]);
 pfPreMultMat(H_matrix, homo);
}

void
rigid_body::update_position(float dt)
{
 pfGetMatRowVec3(H_matrix, 3, location);
}

void
rigid_body::world_to_body(pfVec3 destination, pfVec3 source)
{
 pfMatrix inv_H;
 pfInvertMat(inv_H, H_matrix);
 post_mult(destination, inv_H, source);
}

void
rigid_body::body_to_world(pfVec3 destination, pfVec3 source)
{
 post_mult(destination, H_matrix, source);
}

void
rigid_body::add_force_and_torques(pfVec3 r, pfVec3 f)
{
 pfVec3 t;
 pfCrossVec3(t, r, f);
 pfAddVec3(torques, torques, t);
 pfAddVec3(forces, forces, f);
}
```

```
void
homogeneous_transform(pfMatrix homo,
                      float azimuth, float elevation, float roll,
                      float x,       float y,         float z)
{
 float sz, cz, sy, cy, sx, cx;

 pfSinCos(rad_to_deg(azimuth), &sz, &cz);
 pfSinCos(rad_to_deg(elevation), &sy, &cy);
 pfSinCos(rad_to_deg(roll), &sx, &cx);
 /* col 1 */
 homo[0][0] = cz*cy;
 homo[1][0] = cz*sy*sx - sz*cx;
 homo[2][0] = cz*sy*cx + sz*sx;
 homo[3][0] = x;
 /* col 2 */
 homo[0][1] = sz*cy;
 homo[1][1] = cz*cx + sz*sy*sx;
 homo[2][1] = sz*sy*cx + cz*sx;
 homo[3][1] = y;
 /* col 3 */
 homo[0][2] = -sy;
 homo[1][2] = cy*sx;
 homo[2][2] = cy*cx;
 homo[3][2] = z;
 /* col 4 */
 homo[0][3] = homo[1][3] = homo[2][3] = 0.0f;
 homo[3][3] = 1.0f;
}

void
post_mult(pfVec3 destination, pfMatrix m, pfVec3 source)
{
 destination[0] = source[0] * m[0][0] + source[1] * m[1][0] +
                  source[2] * m[2][0] + m[3][0];

 destination[1] = source[0] * m[0][1] + source[1] * m[1][1] +
                  source[2] * m[2][1] +          m[3][1];

 destination[2] = source[0] * m[0][2] + source[1] * m[1][2] +
                  source[2] * m[2][2] +          m[3][2];
}
```

```
// file "misc.H"

#ifndef _MY_MISC_
#define _MY_MISC_

/* type BOOLEAN */

typedef int boolean;

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define PI  3.14159265358979323846
#define PI_F 3.14159f

/* angle measurement conversions */

float
deg_to_rad(float deg);

float
rad_to_deg(float rad);

double
deg_to_rad(double deg);

double
rad_to_deg(double rad);

#endif
```

```
// file "misc.C"

#include "misc.H"

/* angle measurement conversions */

float
deg_to_rad(float deg)
{
 const float rad_per_deg = PI_F / 180.0f;
 return (deg * rad_per_deg);
}

float
rad_to_deg(float rad)
{
 const float deg_per_rad = 180.0f / PI_F;
 return (rad * deg_per_rad);
}

double
deg_to_rad(double deg)
{
 const double rad_per_deg = PI / 180.0;
 return (deg * rad_per_deg);
}

double
rad_to_deg(double rad)
{
 const double deg_per_rad = 180.0 / PI;
 return (rad * deg_per_rad);
}
```

LIST OF REFERENCES

Chen, C., *Analog and Digital Control System Design: Transfer-Function, State-Space, and Algebraic Methods*, Saunders College Publishing, 1993.

Davidson, S. L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

Frank, A. A., and McGhee, R. B., "Some Considerations Relating to the Design of Autopilots for Legged Vehicles," *Journal of Terramechanics*, Vol. 6, No. 1, pp. 23-35, 1969.

Fu, K. S., Gonzalez, R. C., and Lee, C. S. G., *Robotics: Control, Sensing, Vision and Intelligence*, McGraw-Hill, 1987.

Goetz, J., *Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

Halliday, D., and Resnick, R., *Fundamentals of Physics*, 2d ed., John Wiley & Sons, 1981.

Iwasaki, M., et al., "Development on Aquatic Walking Robot for Underwater Inspection," *Report of the Port and Harbour Reasearch Institute*, Vol. 26, No. 5, pp. 393-422, December 1987.

Koozekanani, S. H., Barin, K., McGhee, R. B., and Chang, H. T., "A Recursive Free-Body Approach to Computer Simulation of Human Postural Dynamics," *IEEE Transactions on Biomedical Engineering*, Vol. BME-30, No. 12, December 1983.

Koschmann, T. D., *The Common LISP Companion*, John Wiley & Sons, 1990.

Kwak, S. H., and McGhee, R. B., "Rule-Based Motion Coordination for a Hexapod Walking Machine," *Advanced Robotics*, Vol. 4, No. 3, pp.263-282, 1990.

McGhee, R. B., "Vehicular Legged Locomotion,", *Advances in Automation and Robotics*, ed. by G. N. Saridis, Vol. 1, Chapter 7, JAI Press Inc., 1985.

McGhee, R. B., Nakano, E., Koyachi, N., and Adachi, H., "An Approach to Computer Coordination of Motion for Energy-Efficient Walking Machines," *Bulletin of Mechanical Engineering Laboratory, Japan*, No. 43, 1986.

Silicon Graphics, Inc., *IRIS Performer Programming Guide*, 1992.

McMillan, S., Orin, D. E., and McGhee, R. B., "Efficient Dynamic Simulation of an Unmanned Underwater Vehicle with a Manipulator," Proceedings of 1994 IEEE International Conference on Robotics and Automation, San Diego, California, May 8-13, 1994.

McPherson, G., *An Introduction to Electrical Machines and Transformers*, John Wiley & Sons, 1981.

Oakley, C. O., *Calculus: A Modern Approach*, Barnes & Noble, 1971.

Schue, A. S., *Simulation of Tripod Gaits for a Hexapod Underwater Walking Machine*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1993.

Waldron, K. J., and McGhee, R. B., "The Adaptive Suspension Vehicle," *IEEE Control Systems Magazine*, Vol. 6, No. 6, December 1986.

Wiener, R. S., and Pinson, L. J., *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, 1988.

Yoneda, K., Suzuki, K., and Kanayama, Y., "Gait and Foot Trajectory Planning for Versatile Motion of a Six Legged Robot," Proceedings of 1994 IEEE International Conference on Robotics and Automation, San Diego, California, May 8-13, 1994.

Yoshikawa, T., *Foundations of Robotics*, MIT Press, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Robert B. McGhee, Code CS/Mz Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
5.	Dr. David R. Pratt, Code CS/Pr Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
6.	Dr. Se-Hung Kwak Loral / ADS 50 Moulton St. Cambridge, MA 02138	1
7.	Dr. Yutaka Kanayama, Code CS/Ka Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
8.	Mr. Hidetoshi Takahashi Port and Harbour Research Institute Ministry of Transport 1-1, 3-Chome, Nagase Yokosuka, Japan	1
9.	Lt. Karl J. R. W. Kristiansen 264 Worden St. Portsmouth, RI 02871	2
10.	Lt. John Goetz R.R. 1, Box 133 Florence, SD 57235	1